

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau(43) International Publication Date
15 August 2002 (15.08.2002)

PCT

(10) International Publication Number
WO 02/063479 A1(51) International Patent Classification⁷: G06F 13/00.
11/32

European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

(21) International Application Number: PCT/US02/03218

(22) International Filing Date: 4 February 2002 (04.02.2002)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/266,185 2 February 2001 (02.02.2001) US
09/777,190 2 February 2001 (02.02.2001) US(71) Applicant (for all designated States except US): DATA-
SYNAPSE, INC. [US/US]; 632 Broadway, New York, NY
10012-2614 (US).

(72) Inventors; and

(75) Inventors/Applicants (for US only): BERNARDIN,
James [—/—]; 225 Park Place, Brooklyn, NY 11238
(US). LEE, Peter [US/US]; 439 East 75th Street, New
York, NY 10021 (US). LEWIS, James [US/US]; 960
Keeler Avenue, Berkeley, CA 94708 (US).(74) Agents: DIMATTEO, John, M. et al.; Patterson, Belknap,
Webb & Tyler, LLP, 1133 Avenue of the Americas, New
York, NY 10036-6710 (US).(81) Designated States (national): AE, AG, AL, AM, AT, AU,
AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU,
CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH,
GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC,
LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW,
MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG,
SI, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ,
VN, YU, ZA, ZM, ZW.(84) Designated States (regional): ARIPO patent (GH, GM,
KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW),
Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM),

Declarations under Rule 4.17:

- as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii)) for the following designations AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZM, ZW, ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG)
- as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii)) for the following designations AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZM, ZW, ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG)

Published:

- with international search report
- before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: DISTRIBUTED COMPUTING SYSTEM

(57) Abstract: The invention provides an off-the-shelf product solution to target the specific needs of commercial users with naturally parallel applications. A top-level, public API provides a simple "compute server" or "task farm" model that dramatically accelerates integration and deployment. By providing built-in, turnkey support for enterprise features like fault-tolerant scheduling, fail-over, load balancing, and remote, central administration, the invention eliminates the need for customized middleware and yields enormous, on-going savings in maintenance and administrative overhead.



WO 02/063479 A1

DISTRIBUTED COMPUTING SYSTEM

FIELD OF THE INVENTION

The present invention relates generally to the field of high-performance computing
5 ("HPC") and, more specifically, to systems and techniques for distributed and/or parallel
processing.

CROSS-REFERENCE TO RELATED APPLICATIONS

This application claims priority from the following co-pending U. S. Patent Applications:
(i) S/N 09/583,244, *Methods, Apparatus, and Articles-of-Manufacture for Network Based*
10 *Distributed Computing*, filed May 31,2000; (ii) S/N 09/711,634, *Methods, Apparatus and*
Articles-of-Manufacture for Providing Always-Live Distributed Computing, filed November
13,2000; (iii) S/N 09/777,190, *Redundancy-Based Methods, Apparatus and Articles-of-*
Manufacture for Providing Improved Quality-of-Service in an Always-Live Distributed
Computing Environment, filed February 2,2001; (iv) S/N 60/266,185, *Methods, Apparatus and*
15 *Articles-of-Manufacture for Network-Based Distributed Computing*, filed February 2, 2001,
now published as WO0188708. Each of the aforementioned co-pending applications (i)- (iv) is
hereby incorporated by reference herein.

BACKGROUND OF THE INVENTION

HPC has long been a focus of both academic research and commercial development, and
20 the field presents a bewildering array of standards, products, tools, and consortia. Any attempt at
comparative analysis is complicated by the fact that many of these interrelate not as mutually
exclusive alternatives, but as complementary component or overlapping standards.

Probably the most familiar, and certainly the oldest, approach is based on dedicated
supercomputing hardware. The earliest supercomputers included vector-based array processors,
25 whose defining feature was the capability to perform numerical operations on very large data
arrays, and other SIMD (Single-Instruction, Multiple-Data) architectures, which essentially
performed an identical sequence of instructions on multiple datasets simultaneously. More
recently, multiple-instruction architectures, and especially SMPs (Symmetric Multi-Processors),
have tended to predominate, although the most powerful supercomputers generally combine
30 features of both.

With dramatic improvements in the processing power and storage capacity of "commodity" hardware and burgeoning network bandwidth, much of the focus has shifted toward parallel computing based on loosely-coupled clusters of general-purpose processors, including clusters of network workstations. Indeed, many of the commercially available high-performance hardware platforms are essentially networks of more or less generic processors with access to shared memory and a high-speed, low latency communications bus. Moreover, many of the available tools and standards for developing parallel code are explicitly designed to present a uniform interface to both multi-processor hardware and network clusters. Despite this blurring around the edges, however, it is convenient to draw a broad dichotomy between conventional hardware and clustering solutions, and the discussion below is structured accordingly.

Conventional hardware solutions

Typical commercial end-users faced with performance bottlenecks consider hardware solutions ranging from mid- to high-end SMP server configurations to true "supercomputers." In practice, they often follow a tortuous, incremental migration path, as they purchase and outgrow successively more powerful hardware solutions.

The most obvious shortcoming of this approach is the visible, direct hardware cost, but even more important are the indirect costs of integration, development, administration, and maintenance. For example, manufacturers and resellers generally provide support at an annual rate equal to approximately 20-30% of the initial hardware cost. Moreover, the increase in physical infrastructure requirements and the administrative burden is much more than linear to the number of CPUs.

But by far the most important issue is that each incremental hardware migration necessitates a major redevelopment effort. Even when the upgrade retains the same operating system (e.g., from one Sun Solaris™ platform to another), most applications require substantial modification to take advantage of the capabilities of the new target architecture. For migrating from one operating system to another (e.g., from NT™ or Solaris™ to Irix™), the redevelopment cost is typically comparable to that of new development, but with the additional burden of establishing and maintaining an alternative development environment, installing and testing new tools, etc. Both development and administration require specialized skill sets and dedicated personnel.

In sum, other indirect costs often total 7 to 9× direct hardware costs, when personnel, time-to-market, and application redevelopment costs are taken into account.

Clusters, grids, and virtual supercomputers

The basic idea of bundling together groups of general-purpose processors to attack large-scale computations has been around for a long time. Practical implementation efforts, primarily within academic computer science departments and government research laboratories, began in earnest in the early 1980s. Among the oldest and most widely recognized of these was the Linda project at Yale University, which resulted in a suite of libraries and tools for distributed parallel processing centered around a distributed, shared memory model.

More elaborate and at a somewhat higher level than Linda, but similar in spirit, PVM (for Parallel Virtual Machine) provided a general mechanism-based on a standard API and messaging protocol for parallel computation over networks of general-purposes processors. More recently, MPI (the Message Passing Interface) has gained ground. Although they differ in many particulars, both are essentially standards that specify an API for developing parallel algorithms and the behavioral requirements for participating processors. By now, libraries provide access to the API from C and/or Fortran. Client implementations are available for nearly every operating system and hardware configuration.

Grid Computing represents a more amorphous and broad-reaching initiative – in certain respects, it is more a philosophical movement than an engineering project. The overarching objective of Grid Computing is to pool together heterogeneous resources of all types (e.g., storage, processors, instruments, displays, etc.), anywhere on the network, and make them available to all users. Key elements of this vision include decentralized control, shared data, and distributed, interactive collaboration.

A third stream of development within high-performance distributed computing is loosely characterized as "clustering." Clusters provide HPC by aggregating commodity, off-the-shelf technology (COTS). By far the most prominent clustering initiative is Beowulf, a loose confederation of researchers and developers focused on clusters of Linux-based PCs. Another widely recognized project is Berkeley NOW (Network of Workstations), which has constructed a distributed supercomputer by linking together a heterogeneous collection of Unix and NT workstations over a high-speed switched network at the University of California.

There is considerable overlap among these approaches. For example, both Grid implementations and clusters frequently employ PVM, MPI, and/or other tools, many of which were developed initially to target dedicated parallel hardware. Nor is the terminology particularly well defined; there is no clear division between "grids" and "clusters," and some authors draw a distinction between "clusters" or dedicated processors, as opposed to "NOWs" (Networks of Workstations), which enlist part-time or intermittently available resources.

Clusters and grids as enterprise solutions

The vast majority of clusters and Grid implementations are deployed within large universities and Government research laboratories. These implementations were specifically developed as alternatives to dedicated supercomputing hardware, to address the kinds of research problems that formed the traditional domain of supercomputing. Consequently, much of the development has focused on emulating some of the more complex features of the parallel hardware that are essential to address these research problems.

The earliest commercial deployments also targeted traditional supercomputing applications. Examples include: hydrodynamics and fluid-flow, optics, and manufacturing process control. In both research and commercial settings, clustering technologies provide at least a partial solution for two of the most serious shortcomings of traditional supercomputing: (1) up-front hardware cost, and (2) chronic software obsolescence (since the system software to support distributed computing over loosely coupled networks must, out of necessity, provide substantial abstraction of the underlying hardware implementation).

However, clusters and grid implementations share, and in many cases, exacerbate, some of the most important weaknesses of supercomputing hardware solutions, particularly within a commercial enterprise environment. Complex, low-level APIs necessitate protracted, costly development and integration efforts. Administration, especially scheduling and management of distributed resources, is burdensome and expensive. In many cases, elaborate custom development is needed to provide fault tolerance and reliability. Both developers and administrators require extensive training and special skills. And although clusters offer some advantages versus dedicated hardware with respect to scale, fragility and administrative complexity effectively impose hard limits on the number of nodes – commercial installations with as many as 50 nodes are rare, and only a handful support more than 100.

These weaknesses have become increasingly apparent as commercial deployments have moved beyond traditional supercomputing applications. Many of the most important commercial applications, including the vast majority of process-intensive financial applications, are "naturally parallel." That is, the computation is readily partitioned into a number of more or less independent sub-computations. Within financial services, the two most common sources of natural parallelism are portfolios, which are partitioned by instrument or counterparty, and simulations, which are partitioned by sample point. For these applications, complex features to support process synchronization, distributed shared memory, and inter-process communication are irrelevant – a basic "compute server" or "task farm" provides the ideal solution. The features that are essential, especially for time-sensitive, business-critical applications, are fault-tolerance, reliability, and ease-of-use. Unnecessary complexity drives up development and administration costs, undermines reliability, and limits scale.

HPC in the financial services industry

The history of HPC within financial services has been characterized by inappropriate technology. One of the earliest supercomputing applications on Wall Street was Monte Carlo valuation of mortgage-backed securities (MBS) – a prototypical example of "naturally parallel" computation. With deep pockets and an overwhelming need for computing power, the MBS trading groups adopted an obvious, well-established solution: supercomputing hardware, specifically MPPs (Massively Parallel Processors).

Although this approach solved the immediate problem, it was enormously inefficient. The MPP hardware that they purchased was developed for research applications with intricate inter-process synchronization and communication requirements, not for naturally parallel applications within a commercial enterprise. Consequently, it came loaded with complex features that were completely irrelevant for the Monte Carlo calculations that the MBS applications required, but failed to provide many of the turnkey administrative and reliability features that are typically associated with enterprise computing. Protracted in-house development efforts focused largely on customized middleware that had nothing to do with the specific application area and resulted in fragile implementations that imposed an enormous administrative burden. Growing portfolios and shrinking spreads continued to increase the demand for computing power, and MPP solutions wouldn't scale, so most of these development efforts have been repeated many times over.

As computing requirements have expanded throughout the enterprise, the same story has played out again and again – fixed-income and equity derivatives desks, global credit and market risk, treasury and Asset-Liability Management (ALM), etc., all have been locked in an accelerating cycle of hardware obsolescence and software redevelopment.

5 More recently, clustering and grid technologies have offered a partial solution, in that they reduce the upfront hardware cost and eliminate some of the redevelopment associated with incremental upgrades. But they continue to suffer from the same basic defect – as an outgrowth of traditional supercomputing, they are loaded with irrelevant features and low-level APIs that drive up cost and complexity, while failing to provide turnkey support for basic enterprise
10 requirements like fault-tolerance and central administration.

The invention, as described below, provides an improved, Grid-like distributed computing system that addresses the practical needs of real-world commercial users, such as those in the financial services and energy industries.

BRIEF SUMMARY OF THE INVENTION

15 The invention provides an off-the-shelf product solution to target the specific needs of commercial users with naturally parallel applications. A top-level, public API provides a simple "compute server" or "task farm" model that dramatically accelerates integration and deployment. By providing built-in, turnkey support for enterprise features like fault-tolerant scheduling, fail-over, load balancing, and remote, central administration, the invention eliminates the need for
20 customized middleware and yields enormous, on-going savings in maintenance and administrative overhead.

Behind the public API is a layered, peer-to-peer (P2P) messaging implementation that provides tremendous flexibility to configure data transport and overcome bottlenecks, and a powerful underlying SDK based on pluggable components and equipped with a run-time XML
25 scripting facility that provides a robust migration path for future enhancements.

Utilizing the techniques described in detail below, the invention supports effectively unlimited scaling over commoditized resource pools, so that end-users can add resources as needed, with no incremental development cost. The invention seamlessly incorporates both dedicated and intermittently idle resources on multiple platforms (Windows™, Unix, Linux,
30 etc.). And it provides true idle detection and automatic fault-tolerant rescheduling, thereby

harnessing discrete pockets of idle capacity without sacrificing guaranteed service levels. (In contrast, previous efforts to harness idle capacity have run low-priority background jobs, restricted utilization to overnight idle periods, or imposed intrusive measures, such as checkpointing.) The invention provides a system that can operate on user desktops during peak
5 business hours without degrading performance or intruding on the user experience in any way.

While the above discussion outlines some of the important features and advantages of the invention, those skilled in the art will recognize that the invention contains numerous other novel features and advantages, as described below in connection with applicants' preferred LiveCluster embodiment.

10 Accordingly, generally speaking, and without intending to be limiting, one aspect of the invention relates to distributed computing systems comprising, for example: a plurality of engines; at least one broker; at least on client application, the client application having an associated driver; the driver being configured to enable communication between the client application and two or more of the engines via a peer-to-peer communication network; the
15 system characterized in that (i) the driver is further configured to enable communication between the client application and the at least one broker over the peer-to-peer network and (ii) the broker is further configured to communicate with the engines over the peer-to-peer network, thereby enabling the broker to control and supervise the execution of tasks provided by the client application on the two or more engines. The system may further include at least
20 one failover broker configured to communicate with the driver and the engines, and, in the event of a broker failure, control and supervise the execution of tasks provided by the client application on the two or more engines. The broker may further include an adaptive scheduler configured to selectively assign and control the execution of tasks provided by the client application on the engines. The adaptive scheduler may be further configured to redundantly
25 assign one or more of the task(s) provided by the client application to multiple engines, so as to ensure the timely completion of such redundantly assigned task(s) by at least one of the engines. The tasks provided by the client application may have associated discriminators. The broker may utilize parameters associated with such discriminators and the engines to determine the assignment of tasks to engines. The system may control the timing of selected

communications between the driver and the engines (or other communications, such as engine-to-engine communications) so as to avoid bottlenecks associated with overloads of the peer-to-peer network, such as delays associated with excessive simultaneous network traffic. The broker and the two or more engines may each include an associated propagator object that

5 permits control over engine-to-engine propagation of data over the peer-to-peer network. The propagator objects may enable an engine or broker node to perform at least three, four, five, six, seven or eight of the following operations: (i) broadcast a message to all nodes, except the current node; (ii) clear all message(s), and associated message.state(s), on specified broker(s) and/or engine(s); (iii) get message(s) for the current node; (iv) get the message(s) from a

10 specified node for the current node; (v) get the state of a specified node; (vi) get the total number of nodes; (vii) send a message to a specified node; and/or (viii) set the state of a specified node.

Still further aspects of the present invention relate to other system configurations, methods, software, encoded articles-of-manufacture and/or electronic data signals comprised of, or produced in accordance with, portions of the preferred LiveCluster embodiment, described in

15 detail below.

BRIEF DESCRIPTION OF THE FIGURES

The present invention will be best appreciated by reference to the following set of figures (to be considered in combination with the associated detailed description) in which:

FIGs. 1-2 depict data flows in the preferred LiveCluster embodiment of the invention;

20 **FIGs. 3-12** are code samples from the preferred LiveCluster embodiment of the invention;

FIG. 13 depicts comparative data flows in connection with the preferred LiveCluster embodiment of the invention;

FIGs. 14-31 are code samples from the preferred LiveCluster embodiment of the

25 invention;

FIG. 32-53 are screen shots from the preferred LiveCluster embodiment of the invention;

FIGs. 33-70 are code samples from the preferred LiveCluster embodiment of the invention;

FIG. 71 illustrates data propagation using propagators in accordance with the preferred

30 LiveCluster embodiment of the inveniton;

FIGs. 72-81 are code samples from the preferred LiveCluster embodiment of the invention; and,

FIGs. 82-87 depict various illustrative configurations of the preferred LiveCluster embodiment of the invention.

5 **DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT(S)**

What follows is a rough glossary of terms used in describing the preferred LiveCluster implementation of the invention.

Broker	A subcomponent of a Server that is responsible for maintaining a "job space," for managing Jobs and Tasks and the associated interactions with Drivers and Engines.
Daemon	A process in Unix that runs in the background and performs specific actions or runs a server with little or no direct interaction. In Windows NT or Windows 2000, these are also called Services.
Director	A subcomponent of a Server that is responsible for routing Drivers and Engines to Brokers.
Driver	The component used to maintain a connection between the LiveCluster Server and the client application.
Engine	The component that actually handles the work of computation, accepting work from and returning results to a Broker.
Failover Broker	A Broker configured to take on work when another Broker fails. The Failover Broker will continue to accept Jobs until another Broker is functioning again, and then it will wait for any remaining Jobs to finish before returning to a wait state.
Job	A unit of work submitted from a Driver to a Server. Servers break apart Jobs into Tasks for further computation.
LiveCluster	LiveCluster provides a flexible platform for distributing large computations to idle, underutilized and/or dedicated processors on any network. The LiveCluster architecture includes a Driver, one or more Servers, and several Engines.
Server	The component of the LiveCluster™ system that takes work from

	Drivers, coordinates it with Engines, and supports Web-based administrative tools. A Server typically contains a Driver and a Broker.
Task	An atomic unit of work. Jobs are broken into Tasks and then distributed to Engines for computation.
Standalone Broker	A Server that has been configured with a Broker, but no Director; its configured primary and secondary Directors are both in other Servers.
Service	A program in Windows NT or Windows 2000 that performs specific functions to support other programs. In Unix, these are also called daemons.

How LiveCluster Works

- LiveCluster supports a simple but powerful model for distributed parallel processing. The basic configuration incorporates three major components — Drivers, Servers, and
- 5 Engines. Generally speaking, the LiveCluster model works as follows:
- A. Client applications (via Drivers) submit messages with work requests to a central Server.
 - B. The Server distributes the work to a network of Engines, or individual CPUs with LiveCluster Installed.
 - 10 C. The Engines return the results to the Server.
 - D. The Server collects the results and returns them to the Drivers.

Tasks and Jobs

- In LiveCluster, work is defined in two different ways: a larger, overall unit, and a smaller piece, or subdivision of that unit. These are called Jobs and Tasks. A **Job** is a unit of
- 15 work. Typically, this refers to one large problem that has a single solution. A Job is split into a number of smaller units, each called a **Task**. An application utilizing LiveCluster submits problems as Jobs, and LiveCluster breaks the Jobs into Tasks. Other computers solve the Tasks and return their results, where they are added, combined, or collated into a solution for the Job.

Component Architecture

The LiveCluster system is implemented almost entirely in Java. Except for background daemons and the installation program, each component is independent of the operating system under which it is installed. The components are designed to support interoperation across both
5 wide and local area networks (WANs and LANs), so the design is very loosely coupled, based on asynchronous, message-driven interactions. Configurable settings govern message encryption and the underlying transport protocol.

In the next section, we describe each of the three major components in the LiveCluster system — Driver, Server, and Engine — in greater detail.

10 Server

The Server is the most complex component in the system. Among other things, the Server:

- Keeps track of the Engines and the ongoing computations (Jobs and Tasks)
- Supports the web-based administration tools — in particular, it embeds a
15 dedicated HTTP Server, which provides the primary administrative interface to the entire system.

Despite its complexity, however, the Server imposes relatively little processing burden. Because Engines and Drivers exchange data directly, so the Server doesn't have to consume a great deal of network bandwidth. By default, LiveCluster is configured so that Drivers and
20 Engines communicate to the Server only for lightweight messages.

The Server functionality is partitioned into two subcomponent entities: the **Broker** and the **Director**. Roughly speaking, the Broker is responsible for maintaining a "job space" for managing Jobs and Tasks and the associated interactions with Drivers and Engines. The primary function of the Director is to manage Brokers. Typically, each Server instance imbeds
25 a Broker/Director pair. The simplest fault-tolerant configuration is obtained by deploying two Broker/Director pairs on separate processors, one as the primary, the other to support failover. For very large-scale deployments, Brokers and Directors are isolated within separate Server instances to form a two-tiered Server network. Ordinarily, in production, the Server is installed as a service (under Windows) or as a daemon (under Unix) — but it can also run
30 "manually," under a log-in shell, which is primarily useful for testing and debugging.

Driver

The **Driver** component maintains the interface between the LiveCluster Server and the client application. The client application code imbeds an instance of the Driver. In Java, the Driver (called JDriver) exists as a set of classes within the Java Virtual Machine (JVM). In
5 C++, the Driver (called Driver++) is purely native, and exists as a set of classes within the application. The client code submits work and administrative commands and retrieves computational results and status information through a simple API, which is available in both Java and C++. Application code can also interact directly with the Server by exchanging XML messages over HTTP.

10 Conceptually, the Driver submits Jobs to the Server, and the Server returns the results of the individual component Tasks asynchronously to the Driver. In the underlying implementation, the Driver may exchange messages directly with the Engines within a transaction space maintained by the Server.

Engine

15 **Engines** report to the Server for work when they are available, accept Tasks, and return the results. Engines are invoked on desktop PCs, workstations, or on dedicated servers by a native daemon. Typically, there will be one Engine invoked per participating CPU. For example, four Engines might be invoked on a four-processor SMP.

An important features of the LiveCluster platform is that it provides reliable
20 computations over networks of interruptible Engines, making it possible to utilize intermittently active resources when they would otherwise remain idle. The Engine launches when it is determined that the computer is idle (or that a sufficient system capacity is available in a multi-CPU setting) and relinquishes the processor immediately in case it is interrupted (for example, by keyboard input on a desktop PC).

25 It is also possible to launch one or more Engines on a given processor deterministically, so they run in competition with other processes (and with one another) as scheduled by the operating system. This mode is useful both for testing and for installing Engines on dedicated processors.

Principles of Operation

Idle Detection

Engines are typically installed on network processors, where they utilize intermittently available processing capacity that would otherwise go unused. This is accomplished by running
5 an extremely lightweight background process on the Engine. This invocation process monitors the operating system and launches an Engine when it detects an appropriate idle condition.

The definition and detection of appropriate idle conditions is inherently platform- and operating-system dependent. For desktop processors, the basic requirement is that the Engine does nothing to interfere with the normal activities of the desktop user. For multi-processor
10 systems, the objective, roughly speaking, is to control the number of active Engines so that they consume only cycles that would otherwise remain idle. In any case, Engines must relinquish the host processor (or their share of it, on multi-processor systems) immediately when it's needed for a primary application. (For example, when the user hits a key on a workstation, or when a batch process starts up on a Server.)

15 Adaptive Scheduling

Fault-tolerant adaptive scheduling provides a simple, elegant mechanism for obtaining reliable computations from networks varying numbers of Engines with different available CPU resources. Engines report to the Server when they are "idle" — that is, when they are available to take work. We say the Engine "logs in," initiating a login session. During the login session,
20 the Engine polls the Server for work, accepts Task definitions and inputs, and returns results. If a computer is no longer idle, the Engine halts, and the task is rescheduled to another Engine. Meanwhile, the Server tracks the status of Tasks that have been submitted to the Engines, and reschedules tasks as needed to ensure that the Job (collection of Tasks) completes.

As a whole, this scheme is called "adaptive" because the scheduling of Tasks on the
25 Engines is demand-driven. So long as the maximum execution time for any Task is small relative to the average "idle window," that is, the length of the average log-in session, between logging in and dropping out, adaptive scheduling provides a robust, scalable solution for load balancing. More capable Engines, or Engines that receive lighter Tasks, simply report more frequently for Work. In case the Engine drops out because of a "clean" interruption — because
30 it detects that the host processor is no longer "idle" — it sends a message to the Server before

it exits, so that the Server can reschedule running Tasks immediately. However, the Server cannot rely on this mechanism alone. In order to maintain performance in the presence of network drop-outs, system crashes, etc., the Server monitors a heartbeat from each active Engine and reschedules promptly in case of time-outs.

5 **Directory Replication**

Directory replication is a method to provide large files that change relatively infrequently. Instead of sending the files each time a Job is submitted and incurring the transfer overhead, the files are sent to each Engine once, where they are cached. The Server monitors a master directory structure and maintains a synchronized replica of this directory on each Engine, by synchronizing each Engine with the files. This method can be used on generic files, or platform-specific items, such as Java .jar files, DLLs, or object libraries.

Basic API Features

Before examining the various features and options provided by LiveCluster, it is appropriate to introduce the basic features of the LiveCluster API by means of several sample programs.

This section discusses the following Java interfaces and classes:

- TaskInput
- TaskOutput
- Tasklet
- 20 • Job
- PropertyDiscriminator
- EngineSession
- StreamJob
- StreamTasklet
- 25 • DataSetJob
- TaskDataSet

The basic LiveCluster API consists of the TaskInput, TaskOutput and Tasklet interfaces, and the Job class. LiveCluster is typically used to run computations on different inputs in parallel. The computation to be run is implemented in a Tasklet. A Tasklet takes a TaskInput, operates on it, and produces a TaskOutput. Using a Job object,

one's program submits `TaskInputs`, executes the job, and processes the `TaskOutputs` as they arrive. The `Job` collaborates with the `Server` to distribute the `Tasklet` and the various `TaskInputs` to `Engines`.

FIG. 1 illustrates the relationships among the basic API elements. Although it is helpful to think of a task as a combination of a `Tasklet` and one `TaskInput`, there is no `Task` class in the API. To understand the basic API better, we will write a simple `LiveCluster` job. The job generates a unique number for each task, which is given to the tasklet as its `TaskInput`. The tasklet uses the number to return a `TaskOutput` consisting of a string. The job prints these strings as it receives them. This is the `LiveCluster` equivalent of a "Hello, World" program. This program will consist of five classes: one each for the `TaskInput`, `TaskOutput`, `Tasklet` and `Job`, and one named `Test` that contains the main method for the program.

TaskInput and TaskOutput

Consider first the `TaskInput` class: The basic API is found in the `com.livecluster.tasklet` package, so one should import that package (see **FIG. 3**). The `TaskInput` interface contains no methods, so one need not implement any. Its only purpose is to mark one's class as a valid `TaskInput`. The `TaskInput` interface also extends the `Serializable` interface of the `java.io` package, which means that all of the class's instance variables must be serializable (or transient). Serialization is used to send the `TaskInput` object from the `Driver` to an `Engine` over the network. As its name suggests, the `SimpleTaskInput` class is quite simple: it holds a single `int` representing the unique identifier for a task. For convenience, one need not make the instance variable private. `TaskOutput`, like `TaskInput`, is an empty interface that extends `Serializable`, so the output class should not be surprising (see **FIG. 4**).

Writing a Tasklet

Now we turn to the `Tasklet` interface, which defines a single method:

```
public TaskOutput service(TaskInput);
```

The `service` method performs the computation to be parallelized. For our Hello program, this involves taking the task identifier out of the `TaskInput` and returning it as part of the `TaskOutput` string (see **FIG. 5**). The `service` method begins by extracting

its task ID from the TaskInput. It then creates a SimpleTaskOutput, sets its instance variable, and returns it. One aspect of the Tasklet interface not seen here is that it, too, extends Serializable. Thus any instance variables of the tasklet must be serializable or transient.

5 With the help of a simple main method (see FIG. 6), one can run this code. This program creates a Tasklet, and then repeatedly creates a TaskInput and calls the Tasklet's service method on it, displaying the results. Although not something one would want to do in practice, this code does illustrate the essential functionality of LiveCluster. In essence, LiveCluster provides a high-performance, fault-tolerant, highly parallel way to
10 repeatedly execute the line:

```
TaskOutput output = tasklet.service(input);
```

The Job Class

To run this code within LiveCluster, one needs a class that extends Job. Recall that a Job is associated with a single tasklet. The needed Job class creates several TaskInputs,
15 starts the job running, and collects the TaskOutputs that result. To write a Job class, one generally writes the following methods:

- (likely) A constructor to accept parameters for the job. It is recommended that the constructor call the setTasklet method to set the job's tasklet.
- (optionally) A createTaskInputs method to create all of the TaskInput
20 objects. Call the addTaskInput method on each TaskInput one creates to add it to the job. Each TaskInput one adds results in one task.
- (required) A processTaskOutput method. It will be called for each TaskOutput that is produced.

The HelloJob class is displayed in FIG. 7. The constructor creates a single
25 HelloTasklet and installs it into the job with the setTasklet method. The createTaskInputs method creates ten instances of SimpleTaskInput, sets their taskIds to unique values, and adds each one to the job with the addTaskInput method. The processTaskOutput method displays the string that is inside its argument.

30

Putting It All Together

The `Test` class (see FIG. 8) consists of a main method that runs the job. The first line creates the job. The second line has to do with distributing the necessary class files to the Engines. The third line executes the job by submitting it to the LiveCluster Server, then waits
5 until the job is finished. (The related `executeInThread` method runs the job in a separate thread, returning immediately.)

The second line of main deserves more comment. First, the `getOptions` method returns a `JobOptions` object. The `JobOptions` class allows one to configure many features of the job. For instance, one can use it to set a name for the job (useful when looking
10 for a job in the Job List of the LiveCluster Administration tool), and to set the job's priority.

Here we use the `JobOptions` method `setJarFile`, which takes the name of a jar file. This jar file should contain all of the files that an Engine needs to run the tasklet. In this case, those are the class files for `SimpleTaskInput`, `SimpleTaskOutput`, and `HelloTasklet`. By calling the `setJarFile` method, one tells LiveCluster to distribute
15 the jar file to all Engines that will work on this job. Although suitable for development, this approach sends the jar file to the Engines each time the job is run, and so should not be used for production. Instead, one should use the file replication service or a shared network file system when in production.

Running the Example

20 Running the above-discussed code will create the following output:

```
    Hello from #0
    Hello from #5
    Hello from #2
    Hello from #4
25    Hello from #9
    Hello from #1
    Hello from #6
    Hello from #7
    Hello from #8
30    Hello from #3
    DONE
```

Summary

- The basic API consists of the `TaskInput`, `TaskOutput` and `Tasklet` interfaces and the `Job` class. Typically, one will write one class that implements `TaskInput`, one that implements `TaskOutput`, one that implements `Tasklet`, and one that extends `Job`.
- 5 • A `Tasklet`'s `service` method implements the computation that is to be performed in parallel. The `service` method takes a `TaskInput` as argument and returns a `TaskOutput`.
- A `Job` object manages a single `Tasklet` and a set of `TaskInputs`. It is responsible for providing the `TaskInputs`, starting the job and processing the
10 `TaskOutputs` as they arrive.
- Some additional code is necessary to create a job, arrange to distribute a jar file of classes, and execute the job.

Data Parallelism

In this section, we will look at a typical financial application: portfolio valuation. Given
15 a portfolio of deals, our program will compute the value of each one. For those unfamiliar with the concepts, a deal here represents any financial instrument, security or contract, such as a stock, bond, option, and so on. The procedure used to calculate the value, or theoretical price, of a deal depends on the type of deal, but typically involves reference to market information like interest rates. Because each deal can be valued independently of the others, there is a
20 natural way to parallelize this problem: compute the value of each deal concurrently. Since the activity is the same for all tasks (pricing a deal) and only the deal changes, we have an example of *data parallelism*. Data-parallel computations are a perfect fit for LiveCluster. A tasklet embodies the common activity, and each `TaskInput` contains a portion of the data.

The Domain Classes

25 Before looking at the LiveCluster classes, we will first discuss the classes related to the application domain. There are six of these: `Deal`, `ZeroCouponBond`, `Valuation`, `DealProvider`, `PricingEnvironment` and `DateUtil`.

Each deal is represented by a unique integer identifier. Deals are retrieved from a database or other data source via the `DealProvider`. `Deal`'s `value` method takes a
30 `PricingEnvironment` as an argument, computes the deal's value, and returns a

Valuation object, which contains the value and the deal ID. ZeroCouponBond represents a type of deal that offers a single, fixed payment at a future time. DateUtil contains a utility function for computing the time between two dates.

The Deal class is abstract, as is its value method (see FIG. 9). The value
5 method's argument is a PricingEnvironment, which has methods for retrieving the interest rates and the valuation date, the reference date from which the valuation is taking place. The value method returns a Valuation, which is simply a pair of deal ID and value. Both Valuation and PricingEnvironment are serializable so they can be transmitted over the network between the Driver and Engines.

10 ZeroCouponBond is a subclass of Deal that computes the value of a bond with no interest, only a principal payment made at a maturity date (see FIG. 10). The value method uses information from the PricingEnvironment to compute the present value of the bond's payment by discounting it by the appropriate interest rate.

The DealProvider class simulates retrieving deals from persistent storage. The
15 getDeal method accepts a deal ID and returns a Deal object. Our version (see FIG. 11) caches deals in a map. If the deal ID is not in the map, a new ZeroCouponBond is created.

With the classes discussed so far, one can write a simple stand-alone application to value some deals (see FIG. 12). This program loads and values 10 deals using a single pricing environment. This LiveCluster application will also take this approach, using the same pricing
20 environment for all deals. The output of this program looks something like:

```
deal ID = 0, value = 3253.5620409955113
deal ID = 1, value = 750.9387692727968
deal ID = 2, value = 8525.835888008573
deal ID = 3, value = 5445.987705373893
25 deal ID = 4, value = 3615.2722123351246
deal ID = 5, value = 1427.1584028651682
deal ID = 6, value = 5824.137556101124
deal ID = 7, value = 2171.6068493160974
deal ID = 8, value = 5099.034037828654
30 deal ID = 9, value = 3652.567194863038
```

With the domain classes finished, we proceed to the LiveCluster application. The basic structure is clear enough: we will have a `ValuationTasklet` class to value deals and return `Valuations`, which will be gathered by a `ValuationJob` class. But there are three important questions we must answer before writing the code:

- 5 1. How are `Deal` objects provided to the tasklet?
2. How is the `PricingEnvironment` object provided to the tasklet?
3. How many deals should a tasklet value at once?

We address the first two of these questions in the next section, "Understanding Data Movement," and the third in the section following, "Understanding Granularity."

10 Understanding Data Movement

The first question is how to provide deals to the tasklet. One choice is to load the deal on the Driver and send the `Deal` object in the `TaskInput`; the other is to send just the deal ID, and let the tasklet load the deal itself. The second way is likely to be much faster, for two reasons: reduced data movement and increased parallelism.

- 15 To understand the first reason, consider FIG. 13, the left portion of which illustrates the connections among the Driver, the Engines, and your data server, on which the deal data resides. The left-hand diagram illustrates the data flow that occurs when the Driver loads deals and transmits them to the Engines. The deal data travels across the network twice: once from the data server to the Driver, and again from the Driver to the Engine. The right-hand diagram
- 20 shows what happens when only the deal IDs are sent to the Engines. The data travels over the network only once, from the data server to the Engine.

The second reason why sending only deal IDs will be faster is that tasklets will try to load deals in parallel. Provided one's data server can keep up with the demand, this can increase the overall throughput of the application.

- 25 These arguments for sending deal IDs instead of deals themselves makes sense for the kind of architecture sketched in FIG. 13, but not for other, less typical configurations. For example, if the Driver and the data server are running on the same machine, then it may make sense, at least from a data movement standpoint, to load the deals in the Driver.

- Let us now turn to the question of how to provide each tasklet with the
- 30 `PricingEnvironment`. Recall that in this application, every deal will be valued with the

same PricingEnvironment, so only a single object needs to be distributed across the LiveCluster. Although the obvious choice is to place the PricingEnvironment in each TaskInput, there is a better way: place the PricingEnvironment within the tasklet itself. The first time that an Engine is given a task from a particular job, it downloads the tasklet object from the Driver, as well as the TaskInput. When given subsequent tasks from the same job, it downloads only the TaskInput, reusing the cached tasklet. So placing an object in the tasklet will never be slower than putting it in a TaskInput, and will be faster if Engines get more than one task from the same job.

One can summarize this section by providing two rules of thumb:

- Let each tasklet load its own data.
- If an object does not vary across tasks, place it within the tasklet.

Understanding Granularity

The third design decision for our illustrative LiveCluster portfolio valuation application concerns how many deals to include in each task. Placing a single deal in each task yields maximum parallelism, but it is unlikely to yield maximum performance. The reason is that there is some communication overhead for each task.

For example, say that one has 100 processors in a LiveCluster, and 1000 deals to price. Assume that it takes 100 ms to compute the value of one deal, and that the total communication overhead of sending a TaskInput to an Engine and receiving its TaskOutput is 500 ms. Since there are 10 times more deals than processors, each processor will receive 10 TaskInputs and produce 10 TaskOutputs during the life of the computation. So the total time for a program that allocates one deal to each TaskInput is roughly $(0.1s \text{ compute time per task} + 0.5s \text{ overhead}) \times 10 = 6 \text{ seconds}$. Compare that with a program that places 10 deals in each TaskInput, which requires only a single round-trip communication to each processor: $(0.1s \times 10) \text{ compute time per task} + 0.5s \text{ overhead} = 1.5 \text{ seconds}$. The second program is much faster because the communication overhead is a smaller fraction of the total computation time. The following table summarizes these calculations, and adds another data point for comparison:

Deals per TaskInput	Elapsed Time
1	6
10	1.5
100	10.5

In general, the granularity—amount of work—of a task should be large compared to the communication overhead. If it is too large, however, then two other factors come into play. First and most obviously, if one has too few tasks, one will not have much parallelism. The third row of the table illustrates this case. By placing 100 deals in each TaskInput, only ten of the 100 available Engines will be working. Second, a task may fail for a variety of reasons—the Engine may encounter hardware, software or network problems, or someone may begin using the machine on which the Engine is running, causing the Engine to stop immediately. When a task fails, it must be rescheduled, and will start from the beginning. Failed tasks waste time, and the longer the task, the more time is wasted. For these reasons, the granularity of a task should not be too large.

Task granularity is an important parameter to keep in mind when tuning an application's performance. We recommend that a task take between one and five minutes. To facilitate tuning, it is wise to make the task granularity a parameter of one's Job class.

15 The LiveCluster Classes

We are at last ready to write the LiveCluster code for our portfolio valuation application. We will need classes for TaskInput, TaskOutput, Tasklet and Job.

The TaskInput will be a list of deal IDs, and the TaskOutput a list of corresponding Valuations. Since both are lists of objects, we can get away with a single class for both TaskInput and TaskOutput. This general-purpose ArrayListTaskIO class contains a single ArrayList (see FIG. 14).

FIG. 15 shows the entire tasklet class. The constructor accepts a PricingEnvironment, which is stored in an instance variable for use by the service method. As discussed above, this is an optimization that can reduce data movement because tasklets are cached on participating Engines.

The `service` method expects an `ArrayListTaskIO` containing a list of deal IDs. It loops over the deal IDs, loading and valuing each deal, just as in our stand-alone application. The resulting `Valuations` are placed in another `ArrayListTaskIO`, which is returned as the tasklet's `TaskOutput`.

5 `ValuationJob` is the largest of the three `LiveCluster` classes. Its constructor takes the total number of deals as well as the number of deals to allocate to each task. In a real application, the first parameter would be replaced by a list of deal IDs, but the second would remain to allow for tuning of task granularity.

10 The `createTaskInputs` method (see FIG. 16) uses the total number of deals and number of deals per task to divide the deals among several `TaskInputs`. The code is subtle and is worth a careful look. In the event that the number of deals per task does not evenly divide the total number of deals, the last `TaskInput` will contain all the remaining deals.

15 The `processTaskOutput` method (see FIG. 17) simply adds the `TaskOutput`'s `ArrayList` of `Valuations` to a master `ArrayList`. Thanks to the deal IDs stored within each `Valuation`, there is no risk of confusion due to `TaskOutputs` arriving out of order.

 The `Test` class has a `main` method that will run the application (see FIG. 18). The initial lines of `main` load the properties file for the valuation application and obtain the values for `totalDeals` and `dealsPerTask`.

 In summary:

- 20 • `LiveCluster` is ideal for data-parallel applications, such as portfolio valuation.
- In typical configurations where the data server and the Driver are on different machines, let each tasklet load its own data from the data server, rather than loading the data into the Driver and distributing it in the `TaskInputs`.
- 25 • Since the `Tasklet` object is serialized and sent to each Engine, it can and should contain data that does not vary from task to task within a job.
- Task granularity—the amount of work that each task performs—is a crucial performance parameter for `LiveCluster`. The right granularity will amortize communication overhead while preventing the loss of too much time due to tasklet failure or interruption. Aim for tasks that run in a few minutes.

30

Engine Properties

In this brief section, we take a look at *Engine properties* in preparation for the next section, on Engine discrimination. Each Engine has its own set of properties. Some properties are set automatically by LiveCluster, such as the operating system that the Engine is running
5 on and the estimated speed of the Engine's processor. Users can also create custom properties for engines by choosing Engine Properties under the Configure section of the LiveCluster Administration Tool.

This chapter also introduces a simple but effective way of debugging tasklets by placing print statements within the `service` method. This output can be viewed from the
10 Administration Tool or written to a log file.

Application Classes

Our exemplary LiveCluster application (see FIG. 19) will simply print out all Engine properties. Since we will not be using `TaskInputs` or generating `TaskOutputs`, we will only need to write classes for the tasklet, `job` and `main` method.

15 The `EnginePropertiesTasklet` class uses LiveCluster's `EngineSession` class to obtain the Engine's properties. It then prints them to the standard output. The method begins by calling `EngineSession`'s `getProperties` method to obtain a `Properties` object containing the Engine's properties. Note that `EngineSession` resides in the `com.livecluster.tasklet.util` package. The tasklet then prints out the list of
20 engine properties to `System.out`, using the convenient `list` method of the `Properties` class.

Where does the output of the `service` method go? Since Engines are designed to run in the background, the output does not go to the screen of the Engine's machine. Instead, it is transmitted to the LiveCluster Server and, optionally, saved to a log file on the Engine's
25 machine. We will see how to view the output in "Running the Program," below.

The `try...catch` is necessary in this method, because `EngineSession.getProperties` may throw an exception and the `service` method cannot propagate a checked exception.

The `EngineSession` class has two other methods, `setProperty` and
30 `removeProperty`, with the obvious meanings. Changes made to the Engine's properties

using these methods will last for the Engine's session. A session begins when an Engine first becomes available and logs on to the Server, and typically ends when the Engine's JVM terminates. (Thus, properties set by a tasklet are likely to remain even after the tasklet's job finishes.) Note that calling the `setProperties` method of the `Properties` object returned from `EngineSession.getProperties` will not change the Engine's properties.

To set an Engine's properties permanently, one should use the Engine Properties tool in the Configure section of the Administration Tool. Click on an Engine in the left column. Then enter property names and values on the resulting page.

10 The `EnginePropertiesJob` class (see FIG. 20) simply adds a few `TaskInputs` in order to generate tasks. `TaskInputs` cannot be null, so empty `TaskInput` object is provided as a placeholder.

The `Test` class is similar to the previously-described `Test` classes.

Running The Program

15 To see what is written to an Engine's `System.out` (or `System.err`) stream, one must open a Remote Engine Log window in the LiveCluster Administration Tool, as follows:

1. From the Manage section of the navigation bar, choose Engine Administration.
2. One should now see a list of Engines that are logged in to one's Server. Click an Engine name in the leftmost column.
- 20 3. One should now see an empty window titled Remote Engine Log. It is important to do these steps *before one* runs the application. By default, Engine output is not saved to a file, so the data sent to this window is transient and cannot be retrieved once the application has completed.

25 The output from each Engine should be similar to that shown in FIG. 21. The meaning of some of these properties is obvious, but others deserve comment. The `cpuNo` property is the number of CPUs in the Engine's computer. The `id` property is unique for each Engine's computer, while multiple Engines running on the same machine are assigned different `instance` properties starting from 0.

30 It is possible to configure an Engine to save its output to a log file as well as sending it to the Remote Engine Log window. One can do this as follows:

1. Visit Engine Configuration in the Configure section of the Administration tool.
2. Choose the configuration one wishes to change from the File list at the top.
3. Find the DSLog argument in the list of properties and set it to true.
4. Click Submit.

- 5 5. When the page reloads, click Save.

The log files will be placed on the Engine's machine under the directory where the Engine was installed. On Windows machines, this is `c:\Program Files\DataSynapse\Engine` by default. In LiveCluster, the log file is stored under `./work/[name] - [instance]/log`.

10 **Summary**

To summarize the above:

- Engine properties describe particular features of each Engine in the LiveCluster.
- Some Engine properties are set automatically; but one can create and set one's own properties in the Engine Properties page of the Administration Tool.
- 15 • The `EngineSession` class provides access to Engine properties from within a tasklet.
- Writing to `System.out` is a simple but effective technique for debugging tasklets. The output goes to the Remote Engine Log window, which can be brought up from Engine Administration in the Administration Tool. One can also configure Engines to
- 20 save the output to a log file.

Discrimination

Discrimination is a powerful feature of LiveCluster that allows one to exert dynamic control over the relationships among Drivers, Brokers and Engines. LiveCluster supports two kinds of discrimination:

- 25 • **Broker Discrimination:** One can specify which Engines and Drivers can log in to a particular Broker. Access this feature by choosing Broker Discrimination in the Configure section of the LiveCluster Administration Tool.
- **Engine Discrimination:** One can specify which Engines can accept a task. This is done in one's code, or in an XML file used to submit the job.

Both kinds of discrimination work by specifying which properties an Engine or Driver must possess in order to be acceptable.

This section discusses only Engine Discrimination, which selects Engines for particular jobs or tasks. Engine Discrimination has many uses. The possibilities include:

- 5 • limiting a job to run on Engines whose usernames come from a specified set, to confine the job to machines under one's jurisdiction;
- limiting a resource-intensive task to run only on Engines whose processors are faster than a certain threshold, or that have more than a specified amount of memory or disk space;
- 10 • directing a task that requires operating-system-specific resources to Engines that run under that operating system;
- inventing one's own properties for Engines and discriminating based on them to achieve any match of Engines to tasks that one desires.

In this section, we will pursue the third of these ideas. We will elaborate our valuation
15 example to include two different types of deals. We will assume that the analytics for one kind of deal have been compiled to a Windows DLL file, and thus can be executed only on Windows computers. The other kind of deal is written in pure Java and therefore can run on any machine. We will segregate tasks by deal type, and use a discriminator to ensure that tasks with Windows-specific deals will be sent only to Engines on Windows machines.

20 Using Discrimination

This discussion will focus on the class `PropertyDiscriminator`. This class uses a `Java Properties` object to determine how to perform the discrimination. The `Properties` object can be created directly in one's code, as we will exemplify below, or can be read from a properties file.

25 When using `PropertyDiscriminator`, one encodes the conditions under which an Engine can take a task by writing properties with a particular syntax. For example, setting the property `cpuMFlops.gt` to the value `80` specifies that the CPU speed of the candidate Engine, in megaflops, must be greater than 80 for the Engine to be eligible.

In general, the discriminator property is of the form
30 *engine_property.operator*. There are operators for string and numerical equality,

numerical comparison, and set membership. They are documented in the Java API documentation for `PropertyDiscriminator`.

Since a single `Properties` object can contain any number of properties, a `PropertyDiscriminator` can specify any number of conditions. All must be true for the
5 Engine to be eligible to accept the task.

In our example, we want to ensure that tasks that contain `OptionDeals` are given only to Engines that run under the Windows operating system. The Engine property denoting the operating system is `os` and its value for Windows is `win32`. So, to construct the right discriminator, one would add the line:

```
10      props.setProperty("os.equals", "win32");
```

to our code.

The Application

Most of the earlier-described classes require no change, including `Deal`,
`ZeroCouponDeal`, `ArrayListTaskIO`, `Valuation`, `PricingEnvironment` and
15 `ValuationTasklet`. We will add another subclass of `Deal`, called `OptionDeal`, whose
`value` method calls the method `nativeValue` to do the work (see FIG. 22).

We assume that the `nativeValue` method is a native method invoking a Windows
DLL. Recall that the `DealProvider` class is responsible for fetching a `Deal` given its
integer identifier. Its `getDeal` method returns either an `OptionDeal` object or
20 `ZeroCouponBond` object, depending on the deal ID it is given. For this example, we decree
that deal IDs less than a certain number indicate `OptionDeals`, and all others are
`ZeroCouponBonds`.

The `ValuationTasklet` class is unchanged, but it is important to note that `Deal`'s
`value` method is now polymorphic:

```
25      output.add(deal.value(_pricingEnvironment));
```

In this line, the heart of `ValuationTasklet`, the call to `value` will cause a Windows
DLL to run if `deal` is an `OptionDeal`.

The `ValuationJob` class has changed significantly, because it must set up the
discriminator and divide the `TaskInputs` into those with `OptionDeals` and those without
30 (see FIG. 23). The first three lines set up a `PropertyDiscriminator` to identify Engines

that run under Windows, as described above. The last two lines call the helper method `createDealInputs`, which aggregates deal IDs into `TaskInputs`, attaching a discriminator. The second argument is the starting deal ID; since deal IDs below `DealProvider.MIN_OPTION_ID` are `OptionDeals`, the above two calls result in the
5 first group of `TaskInputs` consisting solely of `OptionDeals` and the second consisting solely of `ZeroCouponBonds`.

FIG. 24 shows the code for `createDealInputs`. This method takes the number of deals for which to create inputs, the deal identifier of the first deal, and a discriminator. (`IDiscriminator` is the interface that all discriminators must implement.) It uses the same
10 algorithm previously discussed to place Deals into `TaskInputs`. Then calls the two-argument version of `addTaskInput`, passing in the discriminator along with the `TaskInput`.

When `createDealInputs` is invoked to create `OptionDeals`, the `PropertyDiscriminator` we created is passed in. For `ZeroCouponBonds`, the
15 discriminator is `null`, indicating no discrimination is to be done—any Engine can accept the task. Using `null` is the same as calling the one-argument version of `addTaskInput`.

Summary

- Discriminators allow one to control which Engines run which tasks.
- A discriminator compares the properties of an Engine against one or more conditions
20 to determine if the engine is eligible to accept a particular task.
- The `PropertyDiscriminator` class is the easiest way to set up a discriminator. It uses a `Properties` object or file to specify the conditions.
- Discriminators can segregate tasks among Engines based on operating system, CPU speed, memory, or any other property.

25 Streaming Data

The `service` method of a standard `LiveCluster` tasklet uses Java objects for both input and output. These `TaskInput` and `TaskOutput` objects are serialized and transmitted over the network from the Driver to the Engines.

For some applications, it may be more efficient to use streams instead of objects for
30 input and output. For example, applications involving large amounts of data that can process

the data stream as it is being read may benefit from using streams instead of objects. Streams increase concurrency by allowing the receiving machine to process data while the sending machine is still transmitting. They also avoid the memory overhead of deserializing a large object.

- 5 The `StreamTasklet` and `StreamJob` classes enable applications to use streams instead of objects for data transmission.

Application Classes

- Our exemplary application will search a large text file for lines containing a particular string. It will be a parallel version of the Unix `grep` command, but for fixed strings only. Each
10 task is given the string to search for, which we will call the *target*, as well as a portion of the file to search, and outputs all lines that contain the target.

- We will look at the tasklet first. Our `SearchTasklet` class extends the `StreamTasklet` class (see FIG. 25). The `service` method for `StreamTasklet` takes two parameters: an `InputStream` from which it reads data, and an `OutputStream` to
15 which it writes its results (see FIG. 26). The method begins by wrapping those streams in a `BufferedReader` and a `PrintWriter`, for performing line-oriented I/O.

- It then reads its input line by line. If it finds the target string in a line of input, it copies that line to its output. The constructor is given the target, which it stores in an instance variable. Since all tasks will be searching for the same target, the target should be placed in the
20 tasklet. The `service` method is careful to close both its input and output streams when it is finished.

- Users of `StreamTasklet` and `StreamJob` are responsible for closing all streams they are given. Writing a `StreamJob` is similar to writing an ordinary `Job`. One difference is in the creation of task inputs: instead of creating an object and adding it to the job, it obtains
25 a stream, writes to it, and then closes it. The `SearchJob` class's `createTaskInputs` method illustrates this (see FIG. 27; `_linesPerTask` and `_file` are instance variables set in the constructor). The method begins by opening the file to be searched. It writes each group of lines to an `OutputStream` obtained with the `createTaskInput` method. (To generate the input for a task, one calls the `createTaskInput` method, write to the stream
30 it returns, then close that stream.)

The loop within `createTaskInputs` is careful to allocate all of the file's lines to tasks while making sure that no task is given more than the number of lines specified in the constructor.

Like an ordinary `Job`, a `StreamJob` has a `processTaskOutput` method (see
5 **FIG. 28**) that is called with the output of each task. In `StreamJob`, the method's parameter is an `InputStream` instead of a `TaskOutput` object. In this case, the `InputStream` contains lines that match the target. We print them to the standard output. Once again, it is our responsibility to close the stream we are given.

The `Test` class for this example is similar to previous ones.

10 **Improvements**

There are number of ways this basic application can be improved. Let's first consider the final output from the job, the list of matching lines. Because tasks may complete in any order, these lines may not be in their original order within the file. If this is a concern, then line number information can be sent to and returned from the tasklet, and used to sort the
15 matching lines.

If many lines match the target string, then there will be a lot of traffic from the Engines back to the Driver. This traffic can be reduced by returning line numbers, instead of whole lines, from the tasklet. The line numbers can be sorted at the end, and a final pass made over the file to output the corresponding lines. As a further improvement, byte offsets instead of
20 line numbers can be transmitted, enabling the use of random access file I/O to obtain the matching lines from the file. Whether these techniques will in fact result in increased performance will depend on a number of factors, including line length, number of matches, and so on. Experimentation will probably be necessary to find the best design.

Another source of improvement may come from multithreading. `LiveCluster` ensures
25 that calls to `processTaskOutput` are synchronized, so that only one call is active at a time. Thus a naive `processTaskOutput` implementation like the one above will read an entire `InputStream` to completion—a process which may involve considerable network I/O—before moving on to the next. One may achieve better use of the Driver's processor by starting a thread to read the results on each call to `processTaskOutput`.

30 **Summary**

- Use `StreamTasklet` and `StreamJob` when the amount of input or output data is large, and a tasklet can process the data stream as it arrives.
- The `service` method of `StreamTasklet` reads its input from an `InputStream` and writes its results to an `OutputStream`.
- 5 • When writing a `StreamJob` class, create an input for a task by calling the `createTaskInput` method to obtain an `OutputStream`, then writing to and closing that stream.
- The `processTaskOutput` method of `StreamJob` is given an `InputStream` to read a task's results.
- 10 • It is the taskset's responsibility to close all streams.

Data Sets

Although the parallel string search program of the previous section will speed up searching for large files, it misses an opportunity in the case where the same file is searched, over time, for many different targets. As an example of such a situation, consider a web search
15 company that keeps a list of all the questions all users have ever asked so that it can display related questions when a user asks a new one. Although the previous search program will work correctly, it will redistribute the list of previously asked questions to Engines each time a search is done.

A more efficient solution would cache portions of the file to be searched on Engines to
20 avoid repeatedly transmitting it. This is just what LiveCluster's *data set* feature does. A data set is a persistent collection of task inputs (either `TaskInput` objects or streams) that can be used across jobs. The first time it is used, the data set distributes its inputs to Engines in the usual way. But when the data set is used subsequently, it attempts to give a task to an Engine that already has the input for that task stored locally. If all such Engines are unavailable, the
25 task is given to some other available Engine, and the input is retransmitted. Data sets thus provide an important data movement optimization without interfering with LiveCluster's ability to work with dynamically changing resources.

In this section, we will adapt the program of the previous section to use a data set. We will need to use the two classes: `DataSetJob` and `TaskDataSet`. There is no new type of
30 tasklet that we need to consider—as data sets work with existing tasklets.

Using a TaskDataSet

Since a `TaskDataSet` is a persistent object, it must have a name for future reference. One can choose any name:

```
TaskDataSet dataSet = new TaskDataSet("search");
```

- 5 or can call the no-argument constructor, which will assign a name that one can access with the `getName` method.

One can now use the methods `addTaskInput` (for `TaskInput` objects) or `createTaskInput` (for streams) to add inputs to the data set. When finished, call the `doneSubmitting` method:

```
10    dataSet.addTaskInput(t1);  
    dataSet.addTaskInput(t2);  
    dataSet.addTaskInput(t3);  
    dataSet.doneSubmitting();
```

- The data set and its inputs are now stored on the Server and can be used to provide inputs to a
15 `DataSetJob`, as will be illustrated in the next section.

The data set outlives the program that created it. A data set can be retrieved in later runs by using the static `getDataSet` method:

```
TaskDataSet dataSet = TaskDataSet.getDataSet("search");
```

It can be removed with the `destroy` method:

```
20    dataSet.destroy();
```

The Application

- To convert the string search application to use a data set, one must provide a `Job` class that extends `DataSetJob`. To do this, one uses a `DataSetJob` much like an ordinary `Job`, except that instead of providing a `createTaskInputs` method, one provides a data
25 set via the `setTaskDataSet` method (see FIG. 29). The constructor accepts a `TaskDataSet` and sets it into the `Job`. The `processTaskOutput` method of this class is the same as that previously discussed. The `SearchTasklet` class is also the same.

- The main method (see FIG. 30) of the `Test` program creates a `TaskDataSet` and uses it to run several jobs. The method begins by reading a properties file that contains a
30 comma-separated list of target strings, as well as the data file name and number of lines per

task It then creates a data set via the helper method `createDataSetFromFile`. Lastly, it runs several jobs using the data set.

`createDataSetFromFile` (see FIG. 31) places the inputs into a `TaskDataSet`.

Let's review the data movement that occurs when this program is run. When the first job is executed, Engines will pull both the tasklet and a task input stream from the Driver machine. Each engine will cache its stream data on its local disk. When the second and subsequent jobs are executed, the Server will attempt to assign an Engine the same task input that it used for the first job. Then the Engine will only need to download the tasklet, since the Engine has a local copy of the task input.

Earlier, we suggested that if an object does not vary across tasks (but does vary from job to job), it should be placed within the tasklet, rather than inside a task input. Here, we see that idea's biggest payoff. By keeping the task inputs constant, we can amortize their network transmission time over many jobs. Only the relatively small amount of data that varies from job to job—the target string, or in the earlier case, the pricing environment—needs to be transmitted for each new job.

Summary

- Data sets can improve the performance of applications that reuse the same task inputs for many jobs, by reducing the amount of data transmitted over the network.
- A data set is a distributed cache: each Engine has a local copy of a task input. The Server attempts to re-assign a task input to an Engine that had it previously.
- The `TaskDataSet` class allows the programmer to create, retrieve and destroy data sets.
- The `DataSetJob` class extends `Job` to use a `TaskDataSet`.
- Data that varies from job to job should be placed in the tasklet.

25 LiveCluster Administration Tools

The LiveCluster Server provides the LiveCluster Administration Tool, a set of web-based tools that allow the administrator to monitor and manage the Server, its cluster of Engines, and the associated job space. The LiveCluster Administration Tool is accessed from a web-based interface, usable by authorized users from any compatible browser, anywhere on

the network. Administrative user accounts provide password-protected, role-based authorization.

With the screens in the Administration Tool, one can:

- View and modify Server and Engine configuration;
- 5 • Create administrative user accounts and edit user profiles;
- Subscribe to get e-mail notification of events;
- Monitor Engine activity and kill Engines;
- Monitor Job and Task execution and cancel Jobs;
- Install Engines;
- 10 • Edit Engine Tracking properties and change values;
- Configure Broker discrimination;
- View the LiveCluster API, release notes, and other developer documents;
- Download the files necessary to integrate application code and run Drivers;
- View and extract log information;
- 15 • View diagnostic reports; and,
- Run test Jobs.

User Accounts and Administrative Access

All of the administrative screens are password-protected. There is a single "super-user" account, the site administrator, whose hard-coded user name is admin. The site
20 administrator creates new user accounts from the New User screen. Access control is organized according to the five functional areas that appear in the navigation bar. The site administrator is the only user with access to the configuration screens (under Configure), except that each user has access to a single Edit Profile screen to edit his or her own pro-file.

For every other user, the site administrator grants or denies access separately to each of
25 the four remaining areas (Manage, View, Install, and Develop) from the View Users screen. The Server installation script creates a single user account for the site administrator, with both user name and password admin. The site administrator should log in and change the password immediately after the Server is installed.

Navigating the Administration Tool

The administration tools are accessed through the **navigation bar** located on the left side of each screen (see **FIG. 32**). Click one of the links in the navigation bar to display options for that link. Click a link to navigate to the corresponding area of the site. (Note that the navigation bar displays only those areas that are accessible from the current account. If one is not using an administrative account with all privileges enabled, some options will not be visible.) At the bottom of the screen is the **shortcut bar**, containing the **Logout** tool, and shortcut links to other areas, such as **Documentation** and **Product Information**.

The Administration Tool is divided into five sections. Each section contains screens and tools that are explained in more detail in the next five chapters. The following tools are available in each of the sections.

The Configure Section

The **Configure** section contains tools to manage user accounts, profiles, Engines, Brokers, and Directors.

The Manage Section

The **Manage** section enables one to administer Jobs or Tasks that have been submitted, administer data sets or batch jobs, submit a test Job, or retrieve log files.

The View Section

The **View** section contains tools to list and examine Brokers, Engines, Jobs, and data sets. It's different from the **Manage** section in that tools focus on viewing information instead of modifying it, changing configuration, or killing Jobs. One can examine historical values to gauge performance, or troubleshoot one's configuration by watching the interaction between Brokers and Engines interactively.

In general, Lists are similar to the listed displays found in the **Manage** section, which can be refreshed on demand and display more information. Views are graphs implemented in a Java applet that updates in real-time.

The Install Section

The **install** section enables one to install Engines on one's Windows machine, or download the executable files and scripts needed to build installations distributable to Unix machines.

The Develop Section

The Develop section includes downloads and information such as Driver code, API Documentation, Documentation guides, Release Notes, and the Debug Engine.

The Configure Section

The **Configure** section contains tools to manage user accounts, profiles, Engines,
5 Brokers, and Directors. To use any of the following tools, click **Configure** in the Navigation bar to display the list of tools. Then click a tool name to continue.

View/Edit Users

As an administrator, one can change information for existing user accounts. For example, one could change the name of an account, change an account's level of access, or
10 delete an account entirely.

When one clicks **View/Edit Users**, one is presented with a list of defined users, as shown in FIG. 33. To change an existing user account, click the name listed in the **Full Name** column. The display shown in FIG. 34 will open. First, one must enter one's admin password in the top box to make any changes. Then, one can change any of the information for the user
15 displayed. There is also a **Subject** and **Message** section; if one would like to notify the user that changes have been made to his/her account, enter an e-mail message in these fields. To make the change, click **Submit**. One can also delete the account completely by clicking **Delete**. If one would like to create a new user, one must use the **New User Signup** tool.

New User Signup

20 To add a new user, click **New User Signup**. One will be presented with a screen similar to FIG. 34. Enter in one's admin password and the information about the user, and click **Submit**. (Note that the **Subject** and **Message** fields for e-mail notification are already populated with a default message. The placeholders for username and password will be replaced with the actual username and password for the user when the message is sent.)

25 Edit Profile

The **Edit Profile** tool enables you to make changes to the account with which you are currently logged in. It also enables the admin to configure the Server to email notifications of account changes to users. For accounts other than admin, one must click **Edit Profile**, enter one's password in the top box, and make any changes one wishes to make to one's profile.
30 This includes one's first name, last name and email address. One can also change one's

password by entering a new password twice. When one has made the changes, one clicks the **Submit** button. If one is logged in as admin, one can also configure the Server to generate email notifications automatically whenever user accounts are added or modified. To activate this feature, one must provide an email address and the location of the SMTP server. The

5 LiveCluster Server will generate mail from the administrator to the affected users. To disable the email feature, one simply clears the SMTP entry.

Engine Configuration

The Engine Configuration tool (see FIG. 35) enables one to specify properties for each of the Engine types that one deploys. To configure an Engine, one must first choose the Engine

10 type from the **File** list. Then, enter new values for properties in the list, and click **Submit** next to each property to enter these values. Click **Save** to commit all of the values to the Engine configuration. One can also click **Revert** at any time before clicking **Save** to return to the configuration saved in the original file. For more information on any of the properties in the Engine Configuration tool, one can click **Help**.

Engine Properties

This tool (see FIG. 36) displays properties associated with each Engine that has logged in to this Server. A list of Engine IDs is displayed, along with the corresponding Machine Names and properties that are currently assigned to that Engine. These properties are used for discrimination, either in the Broker or the Driver. Properties can be set with this tool, or when

20 an Engine is installed with the 1-Click Install with Tracking link and a tracking profile is created, which is described below, in the **Engine Tracking Editor** tool.

To change the properties assigned to an Engine, one must click the displayed Engine ID in the list. An edit screen (see FIG. 37) is displayed. If there are properties already assigned, one can change their value(s) in an editable box and click **Submit**, or click **Remove** to remove

25 a property completely. To add a new property and value, one may enter them in the editable boxes at the bottom of the list and click **Add**. Once one has finished changing the properties, one may click **Save**. The properties will be sent to the Server, and the Engine will restart. (Note that if Broker discrimination is configured, it is possible to change or add a property that will prevent an Engine from logging back on again.)

Engine Tracking Editor

Engines can be installed with optional tracking parameters, which can be used for discrimination. When Engines are installed with the 1-Click Install with Tracking link, one is prompted for values for these parameters. This tool enables one to define what parameters are given to Engines installed in this method. By default, the parameters include **MachineName**,
5 **Group**, **Location**, and **Description**. One can add more parameters by entering the parameter name in the **Property** column, entering a description of the property type in the **Description** column, and clicking the **Add** button. One can also remove parameters by clicking the **Remove** button next to the parameter one wants to remove.

Broker Configuration

10 The Broker's attributes can be configured by clicking the **Broker Configuration** tool. This displays a hierarchical expanding/collapsing (see FIG. 38) list of all of the attributes of the Broker. One may click on the + and - controls in the left pane to show or hide attributes, or click **Expand All** or **Collapse All** to expand or collapse the entire list.

When one clicks on an attribute, its values are shown in the right pane. One can change
15 an attribute in an editable box by entering a new value and clicking **Submit**. To find more information about each additional attribute, one may click **Help** in the lower right corner of the display. A help window will open with complete details about the attribute.

Broker Discrimination

One can configure Brokers to do discrimination on Engines and Drivers with the
20 **Broker Discrimination** tool (see FIG. 39). First, one must select the Broker one wants to configure from the list at the top of the page. If one is only running a single Broker, there will only be one entry in this list. One can configure discriminators for both Driver properties and Engine properties. For Drivers, a discriminator is set in the Driver properties, and it prevents Tasks from a defined group of Drivers from being taken by this Broker. For Engines and
25 Drivers, discriminators prevent login sessions from being established with a Broker, which changes routing between Brokers and Engines or Drivers.

Each discriminator includes a property, a comparator, and a value. The property is the property defined in the Engine or Driver, such as a group, OS or CPU type. The value can be either a number (double) or string. The comparator compares the property and value. If they
30 are true, the discriminator is matched, and the Engine or Driver can login to a Broker. If they

are false, the Driver can't log in to the Broker, and must use another Broker. In the case of an Engine, it won't be sent Tasks from that Broker. Note that both property names and values are case-sensitive.

One further option for each discriminator is the **Negate other Brokers** box. When this is selected, an Engine or Driver will be considered only for this Broker, and no others. For example, if one has a property named `state` and sets a discriminator for when `state` equals `NY` and selects **Negate other Brokers**, an Engine with `state` set to `NY` will go to this Broker, because other Brokers won't accept its login.

Once one has entered a property, comparator, and value, click **Add**. One can add multiple discriminators to a Broker by defining another discriminator and clicking **Add** again. Click **Save** to save all added discriminators to the Broker. When one saves discriminators, all Engines currently logged in will log out and attempt to log back in. This enables one to set a discriminator to limit a number of Engines and immediately force them to log off.

By default, if an Engine or Driver does not contain the property specified in the discriminator, the discriminator is not evaluated and considered false. However, one can select **Ignore Missing Properties** for both the Driver and Engine. This makes an Engine or Driver missing the property specified in a discriminator ignore the discriminator and continue. For example, if one sets a discriminator for `state = Arizona`, and an Engine doesn't have a `state` property, normally the Broker won't give the Engine Jobs. But if one selects **Ignore Missing Properties**, the Engine without properties will still get Jobs from the Broker.

Director Configuration

To configure the Director, an interface similar to the **Broker Configuration** tool described above is used. When one clicks **Director Configuration**, a hierarchy of attributes is shown, and one can click an attribute to change it. As with the Broker, the Director attributes have a **Help** link available.

Client Diagnostics

If one is troubleshooting issues with one's LiveCluster installation, one can generate and display client statistics using the **Client Diagnostics** tool (see FIG. 40). This generates tables or charts of information based on client messaging times.

To use client diagnostics, one must first select **Client Diagnostics** and then click the **edit diagnostic options** link. Set **Enabled** to true, click **Submit**, then click **Save**. This will enable statistics to be logged as the system runs. (Note that this can generate large amounts of diagnostic data, and it is recommended that one enable this feature only when debugging.)

- 5 Click **diagnostic statistics** to return to the previous screen. Next, one must specify a time range for the analysis. Select a beginning and ending time range, or click **Use all available times** to analyze all information.

After selecting a time range, one can select what data is to be shown, and how it will be shown, either in a table or chart. For the tables, one must select one or more statistic(s) and
10 one or more client(s). For charts, select only one client and one or more statistic for client charts; statistic charts require one to select one statistic and one or more client(s). The table or chart will be displayed in a new window.

Event Subscription

If one has enabled email notifications by entering a **SMTP** address in the admin
15 profile, one can define a list of email addresses, and configure what event notifications are sent to each address with the **Event Subscription** tool (see **FIG. 41**). To enter a subscriber, click **Add a Subscriber**. To change events for a subscriber, click their name in the list. For each subscriber, enter a single email address in the **Email** box. This must be a full email address, in the form `name@your.address.com`. One can enter a string in the **Filter** box to limit
20 notifications to events which contain the string in the event. For example, one could limit notifications to those about an Engine named **Alpha** by entering **Alpha** in the **Filter** box. When the box is left clear (the default), all events are considered for notification.

After specifying an email address and an optional filter, select which events one would like to monitor from the list below. Once one is done, click **Submit**. When each event occurs,
25 the Server will send a short notification message to the specified email address. One can later edit a subscriber's events, filter, or email address by clicking the subscriber's name in the list presented when one selects the **Event Subscription** tool. One can also remove a name completely by clicking the **Remove** button next to it.

The **Manage** section enables one to administer Jobs or Tasks that have been submitted,
30 administer data sets or batch jobs, submit a test Job, or retrieve log files. To use any of the

following tools, click **Manage** in the Navigation bar to display a list of tools at the left. Then click a tool to continue.

Broker Administration

One can view Engines logged on to a Broker, or change the ratio of Engines to Drivers handled by a Broker, by using the **Broker Administration** tool (see FIG. 42). Each Broker logged on to the Director is listed, along with the number of busy and idle Engines logged onto it. Click on the Broker name in the **Hostname** column to display a list of the Engines currently logged in. To see the graphs depicting Broker statistics, click the **Create** button in the Monitor column. One can specify the number of jobs to be displayed in the Broker Monitor by changing the number in the box to the left of the **Create** button. The **Engine Weight** and **Driver Weight** boxes are used to set the ratio of Engines to Drivers that are sent to the Broker from the Director. By default, **Engine Weight** and **Driver Weight** are both 1, so the Broker will handle Engines and Drivers equally. This can also be changed so a Broker favors either Engines or Brokers. For example, changing **Engine Weight** to 10 and leaving **Driver Weight** at 1 will make the Broker handle Engines ten times more than Drivers. To update the list and display the most current information, click the **Refresh** button. One can also automatically update the list by selecting a value from the list next to the **Refresh** button.

Engine Administration

This tool (see FIG. 43) enables one to view and control any Engines currently controlled by one's Server. To update the list and display the most current information, click the **Refresh** button. One can also automatically update the list by selecting a value from the list next to the **Refresh** button.

Engines are displayed by username, with 20 Engines per page by default. One can select a greater number of lists per page, or display all of the Engines, by clicking a number or **All** next to **Results Per Page** on the top right of the screen. One can also find a specific Engine by entering the user-name in the box and clicking **Search For Engines**. The Status column displays if an Engine is available for work. If "Available" is displayed, the Engine is logged on and is ready for work. Engines marked as "Logged off" are no longer available. "Busy" Engines are currently working on a Task. Engines shown as "Logging in" are in the login

process, and are possibly transferring files. One can also click the text in the **Status** column to open a window containing current server logs for that Engine.

To quickly find out more information about an Engine, one may move the mouse over the Engine username without clicking it. A popup window containing statistics will be shown (see FIG. 44). One can also click on an Engine username to display detailed logging on that Engine. If the Engine is currently processing a Job, it is displayed in the **Job-Task** column. Hover the mouse over the entry to display a popup with brief statistics on the Job currently being processed, or click on the entry for a more detailed log. Current Jobs also have their owner displayed in the **Owner** column.

10 **Job Administration**

One can view and administer Jobs posted to a Broker in the **Job Administration** section (see FIG. 45). Here, one is presented with a list of running, completed, and cancelled Jobs on the Broker. To get the most up-to-date information, click the **Refresh** button. One can also automatically refresh the page by selecting an interval from the list next to the **Reload** button.

While a Job is running, one can change its priority by selecting a new value from the list in the **Priority** column. Possible values range from 10, the highest, to 0, the lowest. One can click **Remove Finished Jobs** to display only pending Jobs, vary the number of results per page by clicking on a number, or find a specific Job by searching on its name, similar to the **Engine Administration**.

Jobs are shown in rows with **UserName**, **JobName**, **Submit Time**, **Tasks Completed**, and **Status**. To display information on a Job, point to the Job Name and a popup window containing statistics on the Job appears. For more information, click the Job Name and a graph will be displayed in a new window. One can also click on a Job's status to view its Broker and Director log files. To kill Jobs, select one or more Jobs by clicking the check box in the **Kill** column, or click **Select All** to kill all Jobs, then click **Submit**.

Data Set Administration

Jobs can utilize a **DataSet**, which is a reusable set of **TaskInputs**. Repeated Jobs will result in caching **TaskInputs** on Engines, resulting in less transfer overhead. One can click **Data Set Administration** to view all of the active Data Sets. One can also select Data Sets and

click **Submit** to remove them; however, one will also need to kill the related Jobs. DataSets are usually created and destroyed with the Java API.

Batch Administration

Batch Jobs are items that have been registered with a Server, either by LiveDeveloper, by copying XML into a directory on the Server, or by a Driver. Unlike a Job, they don't immediately enter the queue for processing. Instead, they contain commands, and instructions to specify at what time the tools will execute. These events can remain on the Server and run more than once. Typically, a Batch Job is used to run a Job at a specific time or date, but can be used to run any command.

10 The **Batch Administration** tool (see FIG. 46) displays all Batch Jobs on the Server, and enables one to suspend, resume, or remove them. Each Batch Job is denoted with a name. A **Type** and **Time** specify when the Batch Job will start. If a Batch Job is **Absolute**, it will enter the queue at a given time. A **Relative** Batch Job is defined with a recurring time or a time relative to the current time, such as a Batch Job that runs every hour, or one defined in
15 the cron format. **Immediate** jobs are already in the queue.

To suspend a Batch Job or resume a suspended Batch Job, select it in the **Suspend/Resume** column, and click the **Submit** button below that column. Batch Jobs can be killed by selecting them in the **Remove** column and clicking the **Submit** button below that column, or clicking **Select All** and then **Submit**. Killing a Batch Job does not kill any currently
20 running Jobs that were created by that Batch Job. To kill these, one must use the **Job Administration** tool. Likewise, if one kills a Job from the **Job Administration** tool, one won't kill the Batch Job. For example, if there exists a Batch Job that runs a Job every hour, it is after 4:00, and one kills the Job that appears in the **Job Administration** tool, another Job will appear at 5:00. One must kill both the Job and the Batch Job to stop the Jobs completely.

25 Batch Jobs that are submitted by a Driver will only stay resident until the Server is restarted. To create a Batch Job that will always remain resident, one can create a Batch Job file. To do this, click **new batch file** to open the editor. One can also click the name of a Batch Job that was already created on the Server. One can then enter the XML for the Batch Job, specify a filename, and click **Save** to save the file, **Submit** to enter the file, or **Revert** to
30 abandon the changes.

Test Job

To test a configuration, one can submit a test Job. This tool submits a Job using the standard Linpack benchmark, using an internal Driver. One can set the following parameters for a Linpack test:

Job Name	Name of the Job in the Job Admin.
User Name	Name of the User in the Job Admin.
Tasks	Number of Tasks in the Job.
Priority	Job execution priority, with 10 being the highest, and 0 the lowest.
Duration	Average duration for Tasks in seconds.
Std Dev	Standard deviation of Task duration in percent.
Input Data	Size of Task input data in kilobytes.
Output Data	Size of Task output data in kilobytes.
Compression	Compress input and output data.
Parallel Collection	Start collecting results before all Tasks are submitted.

- 5 After one has set the parameters, one clicks **Submit** to submit the Job. Once the Job is submitted, the Job Administration screen from the Manage section will be displayed. One can then view, update, or kill the Job.

Log Retrieval

- One can display current and historical log information for the Server with the Log Retrieval tool. The interface, *displayed below*, enables one to select a type of log file, a date range, and how one would like to display the log file. To view the current log file, click **Current Server Log**. The current log file is displayed (see FIG. 47), and any new log activity will be continuously added. One can use this feature to watch an ongoing Job's progress, or troubleshoot errors. At any time one is viewing the current log, click **Snapshot** to freeze the current results and open them in a new window. Also, one can click **Clear** to clear the current results. Click **Past Logs** to return to the original display.
- 10
- 15

To view a past log file, first choose what should be included in the file. Select one or more choices: HT Access Log, HT Error Log, Broker Log, Director Log, Broker.xml, Director.xml, Config.xml, and Engine Updates List. One can also click **Select All** to select all of the information. Next, select a date and time that the logs will end, and select the number of hours back from the end time that will be displayed. After one has chosen your data and a range, click one of the **Submit** buttons to display the data. One can choose to display data in the window below, in a new window, or in a zip file. One can also view any zip files you made in the past.

The View Section

10 The **View** section contains tools to list and examine Brokers, Engines, Jobs, and data sets. It's different from the **Manage** section in that tools focus on viewing information instead of modifying it, changing configuration, or killing Jobs. One can examine historical values to gauge performance, or troubleshoot the configuration by watching the interaction between Brokers and Engines interactively. In general, Lists are similar to the listed displays found in 15 the **Manage** section, which can be refreshed on demand and display more information. Views are graphs implemented in a Java applet that updates in real-time. The following tools are available:

Broker List

The **Broker List** tool (see FIG. 48) displays all Brokers currently logged in. It also 20 gives a brief overview of the number of Engines handled by each Broker. To update the list, click the **Refresh** button. One can also automatically update the display by selecting an interval from the list next to the **Refresh** button. Click a Broker's hostname to display its list of Engines. One can also click **Create** to show the **Broker Monitor** graph, described below.

Broker Monitor

25 The **Broker Monitor** tool opens an interactive graph display (see FIG. 49) showing current statistics on a Broker. The top graph is the **Engine Monitor**, a view of the Engines reporting to the Broker, and their statistics over time. The total number of Engines is displayed in green. The employed Engines (Engines currently completing work for the Broker) are displayed in blue, and Engines waiting for work are displayed in red.

The middle graph is the Job View, which displays what Jobs have been submitted, and the number of Tasks completed in each Job. Running Jobs are displayed as blue bars, completed Jobs are grey, and cancelled Jobs are purple. The bottom graph, the Job Monitor, shows the current Job's statistics. Four lines are shown, each depicting Tasks in the Job. They
5 are submitted (green), waiting (red), running (blue), and completed (grey) Tasks. If a newer Job has been submitted since you opened the Broker Monitor, click **load latest job** to display the newest Job.

Engine List

The **Engine List** provides the same information as the **Engine Administration** tool in
10 the **Manage** section, such as Engines and what Jobs they are running. The only difference is the list only allows one to view the Engine list, while the **Engine Administration** tool also has controls that enable one to kill Jobs.

Engine View

The **Engine View** tool opens an interactive graph displaying Engines on the current
15 Broker, similar to the Engine Monitor section of the **Broker Monitor** graph, described above.

Job List

The **Job List** (see FIG. 50) provides the same information as the **Job Administration** tool in the **Manage** section. The only difference is the list only allows one to view Jobs, while the **Job Administration** tool also has controls that enable you to kill Jobs and change their
20 priority.

Data Set List

The **Data Set List** (see FIG. 51) provides the same information as the **Data Set Administration** tool in the **Manage** section. The only difference is the list only allows one to view Data Sets, while the **Data Set Administration** tool also has controls that enable one to
25 make Data Sets unavailable.

Cluster Capacity

The **Cluster Capacity** tool (see FIG. 52) displays the capabilities of Engines reporting to a Server. This includes number of CPUs, last login, CPU speed, free disk space, free memory, and total memory. All Engines, including those not currently online, are displayed.

One may click **Online Engines Only** to view only those Engines currently reporting to the Server, or click **Offline Engines Only** to view Engines that are not currently available.

The Install Section

The install section contains tools used to install Engine on one or more machines.

5 Engine Installation

The install screen (see FIG. 53) enables one to install Engines on a Windows machine, or download the executable files and scripts needed to build installations distributable to Unix machines.

Remote Engine Script

- 10 The remote Engine script is a Perl script written for Unix that enables one to install or start several DataSynapse Engines from a central Server on remote nodes. To use this script, download the file at the **Remote Engine Script** by can holding Shift and clicking the link, or right-click the link and selecting **Save File As....**

The usage of the script is as follows:

- 15 `dslremoteadmin.pl [ACTION] [-f filename] -m MACHINE_NAME -p
PATH_TO_DS] -s server [-n num_engines] [-i ui_idle_wait] [-D
dist_name] [-c min_cpu_busy] [-C max_cpu_busy]`
ACTION can be either install, configure, start, or stop: install installs the
20 DSEngine tree on the remote node and configures the Engine with parameters specified on the
command line listed above; configure configures the Engine with parameters specified on
the command line as as listed above; start starts the remote Engine; and stop stops the
remote Engine.

- One can specify resources either from a file or singularly on the command line using
the -m machine and -p path options. The format of the resource file is: machine_name
25 /path/to/install/dir.

Driver Downloads

The Driver is available in Java and C++ and source code is available for developers to download from this page. One can also obtain the Live Developer suite from this link.

LiveClusterAPI

One can view the LiveClusterAPI by selecting this tool. API documents are available in HTML as generated by JavaDoc for Java and by Doxygen for C++. Also, documentation is available for the LiveClusterXML API, in HTML format.

Documentation

5 This screen contains links to documentation about LiveCluster. Guides are included with the software distribution, in Adobe Acrobat format. To view a guide, click its link to open it. Note: one must have Adobe Acrobat installed to view the guides in pdf format.

Release Notes

10 This link opens a new browser containing notes pertaining to the current and previous releases.

Debug Engine Installation

 A version of the Engine is available to provide debugging information for use with the Java Platform Debugger Architecture, or JPDA. This Engine does not contain the full functionality of the regular Engine, but does provide information for remote debugging via
15 JPDA. One may select this tool to download an archive containing the Debug Engine.

Basic Scheduling

 The Broker is responsible for managing the job space: scheduling Jobs and Tasks on Engines and supervising interactions with Engines and Drivers

Overview

20 Most of the time, the scheduling of Jobs and Tasks on Engines is completely transparent and requires no administration – the "Darwinian" scheduling scheme provides dynamic load balancing and adapts automatically as Engines come and go. However, one needs a basic understanding of how the Broker manages the job space in order to understand the configuration parameters, to tune performance, or to diagnose and resolve problems.

25 Recall that Drivers submit Jobs to the Broker. Each Job consists of one or more Tasks, which may be performed in any order. Conceptually, the Broker maintains a first-in/first-out queue (FIFO) for Tasks within each Job. When the Driver submits the first Task within a Job, the Broker creates a **waiting Task list** for that job, then adds this waiting list to the appropriate **Job list**, according to the Job's priority (see "Job-Based Prioritization," below). Additional
30 Tasks within the Job are appended to the end of the waiting list as they arrive.

Whenever an Engine reports to the Broker to request Work, the Broker first determines which Job should receive service, then assigns the Task at the front of that Job's waiting list to the Engine. (The Engine may not be eligible to take the next Task, however – this is discussed in more detail below.) Once assigned, the Task moves from the **waiting list** to the **pending** list; the pending list contains all the Tasks that have been assigned to Engines. When an Engine completes a task, the Broker searches both the pending and waiting lists. If it finds the Task on either list, it removes it from both, and adds it to the **completed list**. (The Broker may also restart any Engines that are currently processing redundant instances of the same Task. If the Task is not on either list, it was a redundant Task that completed before the Engine restarted, and the Broker ignores it.)

Tasks migrate from the pending list back to the waiting list when the corresponding Engine is interrupted or drops out. In this case, however, the Broker appends the Task to the **front**, rather than the **back**, of the queue, so that Tasks that have been interrupted are rescheduled at a higher priority than other waiting Tasks within the same Job. Also, the Broker can be configured to append redundant instances of Tasks on the pending list to the waiting list; "Redundant Scheduling," below, provides a detailed discussion of this topic.

Discriminators: Task-Specific Engine Eligibility Restrictions

As indicated above, not every Task is eligible to run on every Engine. The `Discriminator` API supports task discrimination based on Engine-specific attributes. In effect, the application code attaches `IDiscriminator` objects to Tasks at runtime to restrict the class of Engines that are eligible to process them. This introduces a slight modification in the procedure described above: When an Engine is ineligible to take a Task, the Broker proceeds to the next Task, and so on, assigning the Engine the first Task it is eligible to take. Note that `Discriminators` establish hard limits; if the Engine doesn't meet the eligibility requirements for any of the Tasks, the Broker will send the Engine away empty-handed, even though Tasks may be waiting.

The Broker tracks a number of predefined properties, such as available memory or disk space, performance rating (megaflops), operating system, and so forth, that the `Discriminator` can use to define eligibility. The site administrator can also establish

additional attributes to be defined as part of the Engine installation, or attach arbitrary properties to Engines "on the fly" from the Broker.

Job-Based Prioritization

Every LiveClusterJob has an associated **priority**. Priorities can take any integer value
5 between zero and ten, so that there are eleven priority levels in all. 0 is the lowest priority, 10 is the highest, and 5 is the default. The LiveClusterAPI provides methods that allow the application code to attach priorities to Jobs at runtime, and priorities can be changed while a Job is running from the LiveClusterAdministration Tool.

When the Driver submits a job at a priority level, it will wait in that priority queue until
10 distributed by the Broker. Two boolean configuration parameters determine the basic operating mode: **Serial Priority Execution** and **Serial Job Execution**. When **Serial Priority Execution** is `true`, the Broker services the priority queues sequentially. That is, the Broker distributes higher priority Jobs, then moves to lower priority Jobs when higher priority Jobs are completed. When **Serial Priority Execution** is `false`, the Broker provides interleaved
15 service, so that lower-priority queues with Jobs will receive some level of service even when higher-priority Jobs are competing for resources. **Serial Job Execution** has similar significance for Jobs of the same priority: When **Serial Job Execution** is `true`, Jobs of the same priority receive strict sequential service; the first Job to arrive is completed before the next begins. When **Serial Job Execution** is `false`, the Broker provides round-robin service to
20 Jobs of the same priority, regardless of arrival time.

The Broker allocates resources among the competing priority queues based on the **Priority Weights** setting. Eleven integer weights determine the relative service rate for each of the eleven priority queues. For example, if the weight for priority 1 is 2, and the weight for priority 4 is 10, the Broker will distribute five priority-4 Tasks for every priority-1 Task
25 whenever Jobs of these two priorities compete. (Priorities with weights less than or equal to zero receive no service when higher priority Tasks are waiting.) The default setting for both **Serial Execution** flags is `false`, and the default setting for the **Priority Weights** scales linearly, ranging from priority 0 at 1, and priority 10 at 11.

It is generally best to leave the flags at their default settings, so that low-priority Tasks
30 don't "starve," and Jobs can't monopolize resources based on time of arrival. Robust solutions

to most resource-contention problems require no more than two or three priority levels, but they do require some planning. In particular, the client application code needs to assign the appropriate priorities to Jobs at runtime, and the priority weights must be tuned to meet minimum service requirements under peak load conditions.

5 **Polling Rates for Engines and Drivers**

In addition to the serial execution flags and the priority weights, there are four remaining parameters under **Job Space** that merit some discussion. These four parameters govern the polling frequencies for Engines and Drivers and the rate at which Drivers upload Tasks to the Server; occasionally, they may require some tuning.

10 Engines constantly poll the Broker when they are available to take work. Likewise, Drivers poll the Broker for results after they submit Jobs. Within each such transaction, the Broker provides the polling entity with a target latency; that is, it tells the Engine or Driver approximately how long to wait before initiating the next transaction.

Total Engine Poll Frequency sets an approximate upper limit on the aggregate rate at
15 which the available Engines poll the Broker for work. The Broker computes a target latency for the individual Engines, based on the number of currently available Engines, so that the total number of Engine polling requests per second is approximately equal to the **Total Engine Poll Frequency**. The integer parameter specifies the target rate in polls per second, with a default setting of 30.

20 The **Result Found / Not Found Wait Time** parameters limit the frequency with which Drivers poll the Server for Job results (TaskOutputs). **Result Found Wait Time** determines approximately how long a Driver waits, after it retrieves some results, before polling the Broker for more, and **Result Not Found Wait Time** determines approximately how long it waits after polling unsuccessfully. Each parameter specifies a target value in milliseconds, and
25 the default settings are 0 and 1000, respectively. That is, the default settings introduce no delay after transactions with results, and a one-second delay after transactions without results.

The **Task Submission Wait Time** limits the rate at which Drivers submit TaskInputs to the Server. Drivers buffer the TaskInput data, and this parameter determines the approximate waiting time between buffers. The integer value specifies the target latency in
30 milliseconds, and the default setting is 0.

The default settings are an appropriate starting point for most intranet deployments, and they may ordinarily be left unchanged. However, these latencies provide the primary mechanism for throttling transaction loads on the Server.

The Task Rescheduler

5 The **Task Rescheduler** addresses the situation in which a handful of Tasks, running on less-capable processors, might significantly delay or prevent Job completion. The basic idea is to launch redundant instances of long-running Tasks. The Broker accepts the first TaskOutput to return and cancels the remaining instances (by terminating and restarting the associated Engines). However, it's important to prevent "runaway" Tasks from consuming
10 unlimited resources and delaying Job completion indefinitely. Therefore, a configurable parameter, **Max Attempts** limits the number of times any given Task will be rescheduled. If a Task fails to complete after the maximum number of retries, the Broker cancels all instances of that Task, removes it from the pending queue, and sends a FatalTaskOutput to the Driver.

15 Three separately configurable strategies govern rescheduling. The three strategies run in parallel, so that tasks are rescheduled whenever one or more of the three corresponding criteria are satisfied. However, none of the rescheduling strategies comes into play for any Job until a certain percentage of Tasks within that Job have completed; the **Strategy Effective Percent** parameter determines this percentage.

20 More precisely, the Driver notifies the Broker when the Job has submitted all its Tasks (from Java or C++, this notification is tied to the return from the createTaskInputs method within the Job class). At that point, the number of Tasks that have been submitted is equal to the total Task count for the Job, and the Broker begins monitoring the number of Tasks that have completed. When the ratio of completed Tasks to the total exceeds the
25 **Strategy Effective Percent**, the rescheduling strategies begin operating.

 The rescheduler scans the pending Task list for each Job at regular intervals, as determined by the **Interval Millis** parameter. Each Job has an associated taskMaxTime, after which Tasks within that Job will be rescheduled. When the strategies are active (based on the **Strategy Effective Per-cent**), the Broker tracks the mean and standard deviation of the
30 (clock) times consumed by each completed Task within the Job. Each of the three strategies

uses one or both of these statistics to define a strategy-specific **time limit** for rescheduling Tasks.

Each time the rescheduler scans the pending list, it checks the elapsed computation time for each pending Task. Initially, rescheduling is driven solely by the `taskMaxTime` for the Job; after enough Tasks complete, and the strategies are active, the rescheduler also compares the elapsed time for each pending Task against the three strategy-specific limits. If any of the limits is exceeded, it adds a redundant instance of the Task to the waiting list. (The Broker will reset the elapsed time for that Task when it gives the redundant instance to an Engine.)

The **Reschedule First** flag determines whether the redundant Task instance is placed at the front of the back of the waiting list; that is, if **Reschedule First** is `true`, rescheduled Tasks are placed at the front of the queue to be distributed before other Tasks that are waiting. The default setting is `false`, which results in less aggressive rescheduling. Thus, the algorithm that determines the threshold for elapsed time, after which Tasks are rescheduled, can be summarized as:

```

15      if (job.completedPercent > strategyEffectivePercent) {
            threshold := min(job.taskMaxTime,
                            percentCompletedStrategy.limit,
                            averageStrategy.limit,
                            standardDevStrategy.limit)
20      } else threshold := job.taskMaxTime

```

Each of the three strategies computes its corresponding limit as follows:

- The **Percent Completed Strategy** returns the maximum long value (effectively infinite, so there is no limit) until the number of waiting Tasks, as a fraction of the total number of Tasks, falls below the **Remaining Task Percent** parameter, after which it returns the mean completion time. In other words, this strategy only comes into play when the Job nears completion (as determined by the **Remaining Task Percent** setting), after which it begins rescheduling every pending Task at regular intervals, based on the average completion time for Tasks within the Job:

```

30      if (percentCompleted < remainingTaskPercent) {
            percentCompletedStrategy.limit := Long.MAX_VALUE

```

```
} else percentCompletedStrategy.limit := mean
```

The default setting for **Remaining Task Percent** is 1, which means that this strategy becomes active after the Job is 99% completed.

- The **Average Strategy** returns the product of the mean completion time and the **Average Limit** parameter (a double). That is, this strategy reschedules Tasks when their elapsed time exceeds some multiple (as determined by the **Average Limit**) of the mean completion time:

```
averageStrategy.limit := averageLimit * mean
```

The default setting for **Average Limit** is 3.0, which means that it reschedules Tasks after they take at least three times as long as average.

- The **Standard Dev Strategy** returns the mean plus the product of the **Standard Dev Limit** parameter (a double) and the standard deviation of the completion times. That is, this strategy reschedules Tasks when their elapsed time exceeds the mean by some multiple (as determined by the **Standard Dev Limit**) of the standard deviation:

```
standardDevStrategy.limit :=  
    mean + (standardDevLimit * standardDeviation)
```

The default setting for **Standard Dev Limit** is 2.0, which means that it reschedules Tasks after they exceed the average by two standard deviations, or in other words, after they've taken longer than about 98% of the completed Tasks.

(Note that if **Reschedule First** is `true`, then Tasks are guaranteed to either complete or fail within **MaxAttempts * MaxTaskTime**.)

Tuning the Rescheduler

Task rescheduling addresses three basic issues:

- It prevents a small number of less capable processors from significantly degrading Job performance and provides fault tolerance and graceful failure when Engine-specific problems prevent Tasks from completing on individual Engines.

- It prevents "runaway" Tasks from consuming unlimited resources and delaying Job completion indefinitely.
- It provides a fail-safe system to insure that all Tasks will complete, despite unexpected problems from other systems.

5 The default settings are reasonable for many environments, but any configuration represents a compromise, and there are some pitfalls to watch out for. In general, aggressive rescheduling is appropriate when there are abundant resources, but with widely differing capabilities. Conversely, to utilize smaller pools of more nearly identical Engines most efficiently, rescheduling should only be configured to occur in exceptional situations.

10 In case this is not possible, it may be necessary to substantially curtail, or even disable, the rescheduling strategies, to prevent repeated rescheduling and ultimately, cancellation, of long-running Tasks. In many cases, it may be possible to reduce the impact of heterogeneous resources by applying discriminators to route long-running Tasks (at least, those that can be identified *a priori*) to more capable processors. (This is generally a good idea in any case,
15 since it smoothes turnaround performance with no loss of efficiency.)

Another approach that can be effective in the presence of abundant resources is simply to increase the **Max Attempts** setting, to allow more rescheduling attempts before a Task is cancelled and returns a `FatalTaskOutput`. Jobs with very few Tasks also work best without rescheduling. For example, with a setting of 40% for **Strategy Effective Percent**, the
20 strategies would become active for a Job with ten Tasks after only four of those Tasks had completed. Therefore, in cases where Jobs have very few Tasks, **Strategy Effective Percent** should be increased. (For example, a setting of 90% ensures that at least nine Tasks complete before launching the strategies, and a setting of 95% requires at least nineteen.)

Finally, note that it is seldom a good idea to disable rescheduling altogether, for
25 example by setting **Max Attempts** to zero. Otherwise, a single incapacitated or compromised Engine can significantly degrade performance or prevent Tasks from completing. Nor should one completely disable the rescheduling strategies without ensuring that every Job is equipped with a reasonable `taskMaxTime`. Without this backstop, runaway application code can permanently remove Engines from service (that is, until an administrator cancels the offending
30 Job manually from the management area on the Server).

The Task Data Set Manager

`TaskDataSet` addresses applications in which a sequence of operations are to be performed on a common input dataset, which is distributed across the Engines. A typical example would be a sequence of risk reports on a common portfolio, with each Engine responsible for processing a subset of the total portfolio.

In terms of the `LiveClusterAPI`, a `TaskDataSet` corresponds to a sequence of Jobs, each of which shares the same collection of `TaskInputs`, but where the `Tasklet` varies from Job to Job. The principal advantage of the `TaskDataSet` is that the scheduler makes a "best effort" to assign each `TaskInput` to the same Engine repeatedly, throughout the session. In other words, whenever possible, Engines are assigned `TaskInputs` that they have processed previously (as part of earlier Jobs within the session). If the `TaskInputs` contain data references, such as primary keys in a database table, the application developer can cache the reference data on an Engine and it will be retained.

The Broker minimizes data transfer by caching the `TaskInputs` on the Engines. The **Task Data Set Manager** plug-in manages the distributed data. When **Cache Type** is set to 0, the Engines cache the `TaskInputs` in memory; when **Cache Type** is set to 1, the Engines cache the `TaskInputs` on the local file system. **Cache Max** and **Cache Percent** set limits for the size of each Engine's cache. **Cache Max** determines an absolute limit, in megabytes. **Cache Percent** establishes a limit as a percentage of the Engine's free memory or disk space (respectively, depending on the setting of **Cache Type**).

The Data Transfer Plug-In

The **Data Transfer** plug-in manages the transfer of `TaskInput` and `Tasklet` objects from the Broker to the Engines and the transfer of `TaskOutput` objects from the Broker to the Drivers. By default, direct data transfer is configured, and the data transfer configuration specified in this plug-in is not used. However, if direct data transfer is disabled, these settings are used. Under the default configuration, the Broker saves the serialized data to disk. When the Broker assigns a Task to an Engine, the Engine picks up the input data at the location specified by the **Base URL**. Similarly, when the Broker notifies a polling Driver that output data is available, the Driver retrieves the data from the location specified by the **Output URL**. Both of these URLs must point to the same directory on the Server, as specified by the

Data Directory. This directory is also used to transfer instructions (the Tasklet definitions) to the Engines. Alternatively, the Broker can be configured to hold the data in memory and accomplish the transfer directly, by enclosing the data within messages. Two flags, **Store Input to Disk** and **Store Output to Disk**, determine which method is used to transfer input data to Engines and output data to Drivers, respectively. (The default setting is `true` in each case; setting the corresponding flag to `false` selects direct transfer from memory.) This default configuration is appropriate for most situations. The incremental performance cost of the round trip to disk and slight additional messaging burden is rarely significant, and saving the serialized Task data to disk reduces memory consumption on the Server. In particular, the direct-transfer mode is feasible only when there is sufficient memory on the Server to accommodate all of the data. Note that in making this determination, it is important to account for peak loads. Running in direct-transfer mode with insufficient memory can result in `java.lang.OutOfMemory-Errors` from the Server process, unpredictable behavior, and severely degraded performance.

15 **The Job Cleaner**

The **Job Cleaner** plug-in is responsible for Job-space housekeeping, such as cleaning up files and state history for Jobs that have been completed or canceled. This plug-in deletes data files associated with Jobs on a regular basis, and cleans the Job Manage and View pages. It uses the **Data Transfer** plug-in to find the data files. If a Job is finished or cancelled, the files are deleted on the next sweep. The plug-in sweeps the Server at regular intervals, as specified by the integer **Attempts Per Day** (the default setting of 2 corresponds to a sweep interval of every 12 hours). The length of time in hours Jobs will remain on the Job Admin page after finished or cancelled is specified by the integer **Expiration Hours**.

The Driver and Engine Managers

25 The **Driver and Engine Managers** play analogous roles for Drivers and Engines, respectively. They maintain the server state for the corresponding client/server connections. The Broker maintains a server-side proxy corresponding to each active session; there is one session corresponding to each Driver and Engine that is logged in.

The Driver Service and Employment Office Plug-Ins

The **Driver Service** plug-in is responsible for the Driver proxies. **Max Number of Proxies** sets an upper limit on the number of Drivers that can log in concurrently. The default value of 100,000, and is typically not modified.

The **Employment Office** plug-in maintains the Engine proxies. In this case, **Max Number of Proxies** is set by the license, and cannot be increased beyond the limit set by the license. (Although it can be set below the limit imposed by the license.)

The Login Managers

Both the Driver and Engine Managers incorporate **Login Managers**. The Login Managers maintain the HTTP connections with corresponding clients (Drivers and Engines), and monitor the heartbeats from active connections for timeouts. User-configurable settings under the **HTTP Connection Managers** include the URL (on the Broker) for the connections, timeout periods for read and write operations, respectively, and the number times a client will retry a read or write operation that times out before giving up and logging a fatal error. The Server install script configures the URL settings, and ordinarily, they should never be modified thereafter. The read/write timeout parameters are in seconds; their default values are 10 and 60, respectively. (Read operations for large blocks of data are generally accomplished by directdownloads from file, whereas uploads may utilize the connection, so the write timeout may be substantially longer.) The default retry limit is 3. These default settings are generally appropriate for most operating scenarios; they may, however, require some tuning for optimal performance, particularly in the presence of unusually large datasets or suboptimal network conditions.

The **Driver and Engine Monitors** track heartbeats from each active Driver and Engine, respectively, and ends connections to Drivers and Engines which no longer respond. The **Checks Per Minute** parameters within each plug-in determine the frequency with which the corresponding monitor sweeps its list of active clients for connection timeouts. Within each monitor, the heartbeat plug-in determines the approximate target rate at which the corresponding clients (Drivers or Engines) send heartbeats to the Broker, and set the timeout period on the Broker as a multiple of the target rate. That is, the timeout period in milliseconds (which is displayed in the browser as well) is computed as the product of the **Max Millis Per Heartbeat** and the **Timeout Factor**. (It may be worth noting that the actual latencies for

individual heartbeats vary randomly between the target maximum and 2/3 of this value; this randomization is essential to prevent ringing for large clusters.)

The default setting for each maximum heartbeat period is 30,000 (30 seconds) and for each timeout factor, 3, so that the default timeout period for both Drivers and Engines is 90 seconds. By default, the Broker Manager checks for timeouts 10 times per minute, while the Engine Manager sweeps 4 times per minute. (Typically, there are many more Engines than Drivers, and Engine outages have a more immediate impact on application performance.)

Other Manager Components

The **Engine File Update Server** manages file updates on the Engines, including both the DataSynapse Engine code and configuration itself, and user files that are distributed via the directory replication mechanism.

The Native Job Adapter

The **Native Job Adapter** provides services to support applications that utilize the C++ or XML APIs. The basic idea is that the Broker maintains a "pseudo Driver" corresponding to each C++ or XML Job, to track the connection state and perform some of the functions that would otherwise be performed by the Java Driver.

The **Result Found** and **Result Not Found Wait Times** have the same significance as the corresponding settings in the **Job Space** plug-in, except that they apply only to the pseudo Drivers. The **Base URL** for connections with native Jobs is set by the install script, and should ordinarily never change thereafter.

The other settings within the **Native Job Adapter** plug-in govern logging for the **Native Bridge Library**, which is responsible for loading the native Driver on each Engine: a switch to turn logging on and off, the log level (1 for the minimum, 5 for the maximum), the name of the log file (which is placed within the Engine directory on each Engine that processes a native Task), and the maximum log size (after which the log rolls over). By default, logging for the Native Bridge is disabled.

The **Native Job Store** plug-in comes into play for native Jobs that maintain persistence of Task-Outputs on the Broker. (Currently, these include Jobs that set a positive value for `hoursTo-KeepData` or are submitted via the `JobSubmitter` class.) The **Data Directory** is the directory in the Broker's local file system where the `TaskOutputs` are

stored; this directory is set by the install script, and should ordinarily not be changed. The **Attempts Per Day** setting determines the number of times per day that the Broker sweeps the data directory for **TaskOutputs** that are no longer needed; the default setting is 24 (hourly).

Utilities

5 The **Utilities** plug-in maintains several administrative functions. The **Revision Information** plug-in provides read-only access to the revision level and build date for each component associated with the Broker. The **License** plug-in, together with its **License Viewer** component, provides similar access to the license settings.

10 The **Log File** plug-in maintains the primary log file for the Broker itself. Settings are available to determine whether log messages are written to file or only to the standard output and error streams, the location of the log file, whether to log debug information or errors only, the log level (when debug messages are enabled), the maximum length of the log file before it rolls over, and whether or not to include stack traces with error messages.

15 The **Mail Server** generates mail notifications for various events on the Broker. The SMTP host can be set here, or from the **Edit Profile** screen for the site administrator. (If this field is blank or "not set," mail generation is disabled.)

20 The **Garbage Collector** monitors memory consumption on the Broker and forces garbage collection whenever the free memory falls below a threshold percentage of the total available memory on the host. Configuration settings are available to determine the threshold percentage (the default value is 20%) and the frequency of the checks (the default is once per minute).

25 The remaining utility plug-ins are responsible for cleaning up log and other temporary files on the Broker. Each specifies a directory or directories to sweep, the sweep frequency (per day), and the number of hours that each file should be maintained before it is deleted. There are also settings to determine whether or not the sweep should recurse through subdirectories and whether to clean out all pre-existing files on startup. Ordinarily, the only user modification to these settings might be to vary the sweep rate and expiration period during testing.

Directory Replication and Synchronization

30 **Mechanism Overview**

The LiveClustersystem provides a simple, easy-to-use mechanism for distributing dynamic libraries (.dll or .so), Java class archives (.jar), or large data files that change relatively infrequently. The basic idea is to place the files to be distributed within a reserved directory on the Server. The system maintains a synchronized replica of the reserved directory structure for each Engine. Updates can be automatically made, or manually triggered. Also, an Engine file update watchdog can be configured to ensure updates only happen when the Broker is idle.

Server-side directory locations

A directory system resides on the Server in which you can put files that will be mirrored to the Engines. The location of these directories is outlined below.

Server-side directories for Windows

Server-side directories are located in the Server install location (usually c:\DataSynapse\Server) plus \livecluster\public_html\updates. Within that directory are two directories: datasynapse and resources. The datasynapse directory contains the actual code for the Engine and support binaries for each platform. The resources directory contains four directories: shared, win32, solaris, and linux. This shared directory is mirrored to all Engine types, and the other three are mirrored to Engines running the corresponding operating system.

Server-side directories for Unix

For Servers installed under Unix, the structure is identical, but the location is the installation directory (usually /opt/datasynapse) plus /Server/Broker/public_html/updates/resources. The directories are also shared, win32, solaris, and linux.

Engine-side directory locations

A similar directory structure resides in each Engine installation. This is where the files are mirrored. The locations are described below.

Engine-side directories for Windows

The corresponding Engine-side directory is located under the root directory for the Engine installation. The default location on Windows is: C:\Program

Files\DataSynapse\Engine\resources and contains the replicated directories shared and win32.

Engine-side directories for Unix

The corresponding Engine-side directory on Unix is the Engine install directory (for example, /usr/local) plus /DSEngine/resources and contains the replicated directories shared and linux for Linux Engines or solaris for Solaris Engines.

Configuring directory replication

The system can be configured to trigger updates of the replicas in one of two modes:

- **Automatic update mode.** The Server continuously polls the file signatures within the designated subdirectories and triggers Engine updates whenever it detects changes; to update the Engines, the system administrator need only add or overwrite files within the directories.
- **Manual update mode.** The administrator ensures that the correct files are located in the designated subdirectories and triggers the updates manually by issuing the appropriate tools through the Administration tool.

Configuring automatic directory updates

1. In the **Configure** section of the Administration tool, select the **Broker Configuration** tool.

2. Click **Engine Manager**, then select **Engine File Update Server**.
3. Set the value of **Enabled** to true.

Once this is set, files added or overwritten within the Server resources directory hierarchy will automatically update on the Engines. The value of **Minutes Per Check** determines the interval at which the Server polls the directory for changes

Manually Updating files

- To update all files to the Engines manually, set **Update Now** to true, and click **Submit**. This triggers the actual transfer of files from the Server to the Engines, and returns the value of **Update Now** to false.

The Engine File Update Watchdog

- By default, the Broker is configured so updates to the Engine files will only happen when the Broker is idle. The Engine file update watchdog provides this function when enabled,

and ensures that all Engines have the same files. When enabled, the watchdog ensures that Engine files are not updated unless there are no Jobs in progress. If a file update is requested (either automatically or manually), the watchdog does not allow any new Jobs to start, and waits for currently running Jobs to complete. When no Jobs are running or waiting, the update will occur.

If the running Jobs don't complete within the specified update period (the default is 60 minutes), the update will not happen, and Jobs will once again be allowed to start. If this happens, one can either try to trigger an update again, specify a longer update period, or try to manually remove Jobs or stop sending new Jobs. When there is a pending update, a notice will be displayed at the top of the Administration Tool. Also, an email notification is sent on update requests, completions, and timeouts if one subscribes to the `FileUpdateEvent` with the Event Subscription tool.

Using Engines with shared network directories

Instead of using directory replication, one can also provide Engines with common files with a shared network directory, such as an NFS mounted directory. To do this, simply provide a directory on a shared server that can be accessed from all of the Engines. Then, go to the **Configure** section of the Administration tool, select **Engine Configuration**, and change the Class directory to point to the shared directory. When one updates the files on the shared server, all of the Engines will be able to use the same files.

CPU Scheduling for Unix

Unix Engines provide the ability to tune scheduling for multi-CPU platforms. This section explains the basic theory of Engine distribution on multi-CPU machines, and how one can configure CPU scheduling to run an optimal number of Engines per machine.

A feature of LiveClusteris that Engines completing work on PCs can be configured to avoid conflicts with regular use of the machine. By configuring an Engine, one can specify at what point other tasks take greater importance, and when a machine is considered idle and ready to take on work. This is called adaptive scheduling, and can be configured to adapt to one's computing environment, be it an office of PCs or a cluster of dedicated servers.

With a single-CPU computer, it's easy to determine when this work state takes place.

For example, using the Unix Engine, one can specify a minimum and maximum CPU

threshold, using the `-c` and `-C` switches when running the `configure.sh` Engine installation script. When non-Engine CPU utilization crosses below the minimum threshold, an Engine is allowed to run; when the maximum CPU usage on the machine is reached, the Engine exits and any Jobs it was processing are rescheduled.

- 5 With a multi-CPU machine, the processing power is best utilized if an Engine is run on each processor. However, determining a machine's collective available capacity isn't as straightforward as with a single-CPU system. Because of this, Unix Engines have two types of CPU scheduling available to determine how Engines behave with multiprocessor systems.

Nonincremental Scheduling

- 10 The simple form of CPU scheduling is called absolute, or nonincremental scheduling. In this method, minimum and maximum CPU utilization refers to the total system CPU utilization, and not individual CPU utilization. This total CPU utilization percentage is calculated by adding the CPU utilization for each CPU and dividing by the number of CPUs. For example, if a four-CPU computer has one CPU running at 50% utilization and the other
15 three CPUs are idle, the total utilization for the computer is 12.5%.

- With nonincremental scheduling, a minimum CPU and maximum CPU are configured, but they refer to the total utilization. Also, they simultaneously apply to all Engines. So if the maximum CPU threshold is set at 25% on a four-CPU machine and four Engines are running, and a non-Engine program pushes the utilization of one CPU to 100%, all four Engines will
20 exit. Note that even if the other three CPUs are idle, their Engines will still exit. In this example, if the minimum CPU threshold was set at 5%, all four Engines would restart when total utilization was below 5%. By default, the Unix Engine uses nonincremental scheduling. Also, Windows Engines always use this method.

Incremental Scheduling

- 25 Incremental scheduling is an alternate method implemented in Unix Engines to provide better scheduling of when Engines can run on multi-CPU computers. To configure incremental scheduling, use the `-I` switch when running the `configure.sh` Engine installation script. With incremental scheduling, minimum CPU and maximum CPU utilization refers to each CPU. For example, if there is an Engine running on each CPU of a multi-CPU system, and the
30 maximum CPU threshold is set at 80%, and a non-Engine program raises CPU utilization

above 80% on that CPU, that Engine will exit, and other Engines will continue to run until their CPU reaches the maximum utilization threshold. Also, an Engine would restart on that CPU when that CPU's utilization dropped below the minimum CPU utilization threshold.

The CPU scheduler takes the minimum and maximum per/CPU settings specified at Engine installation and normalizes the values relative to total system utilization. When these boundaries are crossed, and Engine is started or shut down and the boundaries are recalculated to reflect the change in running processes. This algorithm is used because, for example, a 50 % total CPU load on an eight processor system is typically due to four processes each using 100% of an individual CPU, rather than sixteen processes each using 25 % of a CPU.

The normalized values are calculated with the following assumptions:

1. System processes will be scheduled such that a single CPU is at maximum load before other CPUs are utilized.
2. For computing maximum thresholds, CPUs which do not have Engines running on them are taken to run at maximum capacity before usage encroaches onto a CPU being used by an Engine.
3. For computing minimum thresholds, CPUs which do not have Engines running on them are taken to be running at least the per/CPU maximum threshold.

The normalized utilization of the computer is calculated by the following formulas. The maximum normalized utilization (U_{nmax}) equals:

$$U_{nmax} = \frac{U_{max}}{C_t} + \frac{U_{tot}}{C_t} [C_t - C_r]$$

Where U_{max} = Per-CPU maximum (user specified);

U_{tot} = Maximum value for CPU utilization (always 100);

C_t = Total number of CPUs; and,

C_r = Number of CPUs running Engines.

The minimum normalized utilization (U_{nmin}) equals:

$$U_{nmin} = \frac{U_{min}}{C_t} + \frac{U_{max}}{C_t} [C_t - C_r - 1]$$

The variables are the same as above, with the addition of U_{min} = per-CPU minimum.

The LiveCluster API

The **LiveCluster API** is available in both C++, called **Driver++**, and Java, called **JDriver**. There is also an **XML** facility that can be used to configure or script Java-based Job implementations.

- 5 The **Tasklet** is analogous with the **Servlet** interface, part of the Enterprise Java Platform. For example, a **Servlet** handles web requests, and returns dynamic content to the web user. Similarly, a **Tasklet** handles a task request given by a **TaskInput**, and returns the completed task with **TaskOutput**.

- 10 The three Java interfaces (**TaskInput**, **TaskOutput**, and **Tasklet**) have corresponding pure abstract classes in C++. There is also one partially implemented class, with several abstract/virtual methods for the developer to define, called **Job**.

The C++ API also introduces one additional class, **Serializable**, to support serialization of the C++ Task objects.

How It Works

- 15 To write an application using **LiveCluster**, one's application should organize the computing problem into units of work, or **Jobs**. Each **Job** will be submitted from the **Driver** to the **Server**. To create a **Job**, the following steps take place:

1. Each **Job** is associated with an instance of **Tasklet**.
2. One **TaskOutput** is added to the **Job** to collect results.
- 20 3. The unit of work represented by the **Job** is divided into **Tasks**. For each **Task**, a **TaskInput** is added to the **Job**.
4. Each **TaskInput** is given as input to a **Tasklet** running on an **Engine**. The result is returned to a **TaskOutput**. Each **TaskOutput** is returned to the **Job**, where it is processed, stored, or otherwise used by the application.

- 25 All other handling of the **Job** space, **Engines**, and other parts of the system are handled by the **Server**. The only classes one's program must implement are the **Job**, **Tasklet**, **TaskletInput**, and **TaskletOutput**. This section discusses each of these interfaces, and the corresponding C++ classes.

TaskInput

`TaskInput` is a marker that represents all of the input data and context information specific to a Task. In Java, `TaskInput` extends the `java.io.Serializable` interface:

```
public interface TaskInput extends java.io.Serializable { }.
```

In C++, `TaskInput` extends the class `Serializable`, so it must define methods to read

5 and write from a stream (this is discussed in more detail below):

```
class TaskInput : public Serializable {
    public: virtual ~TaskInput() {}
};
```

The examples show a Monte Carlo approach to calculating Pi (see FIGs. 54-55).

10 **TaskOutput**

`TaskOutput` is a marker that represents all of the output data and status information produced by the Task. (See FIGs. 56-57.)

Like `TaskInput`, `TaskOutput` extends the `java.io.Serializable` interface:

```
15 public interface TaskOutput extends java.io.Serializable { }
```

Similarly, the C++ version extends the class `Serializable`, so it must define methods to read and write from a stream:

```
class TaskOutput : public Serializable {
    public:
20 virtual ~TaskOutput() {}
};
```

Tasklet

The `Tasklet` defines the work to be done on the remote Engines. (See FIGs. 58 and 59A-B.) There is one command-style method, `service`, that must be implemented.

25 Like `TaskInput` and `TaskOutput`, the Java `Tasklet` extends `java.io.Serializable`. This means that the `Tasklet` objects may contain one-time initialization data, which need only be transferred to each Engine once to support many `Tasklets` from the same Job. (The relationship between `Tasklets` and `TaskInput/TaskOutput` pairs is one-to-many.) In particular, for maximum efficiency,

30 shared input data that is common to every task invocation should be placed in the `Tasklet`, and only data that varies across invocations should be placed in the `TaskInputs`.

As above, the Java implementation requires a default constructor, and any non-transient fields must themselves be serializable:

```

    public interface Tasklet extends java.io.Serializable {
        public TaskOutput service(TaskInput input);
5      }

```

The C++ version is equivalent. It extends the class `Serializable`, so it must define methods to read and write from a stream:

```

    class Tasklet : public Serializable {
        public:
10      virtual TaskOutput* service(TaskInput* input) = 0;
        virtual ~Tasklet() { }
    };

```

Job

A **Job** is simply a collection of Tasks. One must implement three methods:

```

15      createTaskInputs
        processTaskOutput
        processFatalOutput

```

(C++ implementations require another method, `getLibraryName`, which specifies the library that contains the `Tasklet` implementation to be shipped to the remote Engines.)

20 Implementations of `createTaskInputs` call `addTaskInput` to add Tasks to the queue. (See FIGs. 60-61.) In addition, **Job** defines static methods for instantiating **Job** objects based on XML configuration scripts and call-backs to notify the application code when the **Job** is completed or encounters a fatal error. A **Job** also implements `processTaskOutput` to read output from each Task and output, process, store, add, or
 25 otherwise utilize the results. Both the C++ and Java versions provide both blocking (`execute`) and non-blocking (`executeInThread`) job execution methods, and `executeLocally` to run the job in the current process. This last function is useful for debugging prior to deployment.

JobOptions

30 Each **Job** is equipped with a **JobOptions** object, which contains various parameter settings. The `getOptions` method of the **Job** class can be used to get or set options in the **JobOptions** object for that **Job**. A complete list of all methods available for the

`JobOptions` object is available in the API reference documentation. Some commonly used methods include `setJobname`, `setJarFile`, and `setDiscriminator`.

setJobname

By default, the name associated with a `Job` and displayed in the Administration Tool is a long containing a unique number. One can set a name that will also be displayed in the Administration Tool with the Job ID. For example, if one's `Job` is named `job`, add this code:

```
job.getOptions().setJobname("Job Number 9");
```

setJarFile

A difference between the C++ and Java versions of the Driver API has to do with the mechanism for distributing code to the Engines.

For both APIs, the favored mechanism of code distribution involves distributing the Jar file containing the concrete class definitions to the Engines using the directory replication mechanism. The C++ version supports this mechanism. The dynamic library containing the implementation of the concrete classes must be distributed to the Engines using the native code distribution mechanism, and the corresponding `Job` implementation must define `getLibraryName` to specify the name of this library, for example `picalc` (for `picalc.dll` on Win32 or `libpicalc.so` on Unix).

With Java, a second method is also available, which can be used during development. The other method of distributing concrete implementations for the `Tasklet`, `TaskInput`, and `TaskOutput` is to package them in a Jar file, which is typically placed in the working directory of the Driver application. In this case, the corresponding `Job` implementation calls `setJarFile` with the name of this Jar file prior to calling one of the `execute` methods, and the Engines pull down a serialized copy of the file when they begin work on the corresponding Task. This method requires the Engine to download the classes each time a `Job` is run.

setDiscriminator

A discriminator is a method of controlling what Engines accept a Task. FIG. 76 contains sample code that sets a simple property discriminator.

Additional C++ Classes

30 Serializable

The C++ API incorporates a class `Serializable`, since object serialization is not a built-in feature of the C++ language. This class (see FIG. 62) provides the mechanism by which the C++ application code and the `LiveClusterMiddleware` exchange object data. It contains two pure virtual methods that must be implemented in any class that derives from it (i.e., in `TaskInput`, `TaskOutput`, and `Tasklet`).

API Extensions

The `LiveClusterAPI` contains several extensions to classes, providing specialized methods of handling data. These extensions can be used in special cases to improve performance or enable access to information in a database.

10 `DataSetJob` and `TaskDataSet`

A `TaskDataSet` is a collection of `TaskInputs` that persist on the Server as the input for any subsequent `DataSetJob`. The `TaskInputs` get cached on the Engine for subsequent use for the `TaskDataSet`. This API is therefore appropriate for doing repeated calculations or queries on large datasets. All Jobs using the same `DataSetJob` will all use the `TaskInputs` added to the `TaskDataSet`, even though their `Tasklets` may differ.

Also, `TaskInputs` from a set are cached on Engines. Engines which request a task from a Job will first be asked to use input that already exists in its cache. If it has no input in its cache, or if other Engines have already taken input in its cache, it will download a new input, and cache it.

20 An ideal use of `TaskDataSet` would be when running many Jobs on a very large dataset. Normally, one would create `TaskInputs` with a new copy of the large dataset for each Job, and then send this large `TaskInputs` to Engines and incur a large amount of transfer overhead each time another Job is run. Instead, the `TaskDataSet` can be created once, like a database of `TaskInputs`. Then, small `Tasklets` can be created that use the `TaskDataSet` for input, like a query on a database. As more jobs are run on this session, the inputs become cached among more Engines, increasing performance.

Creating a `TaskDataSet`

To create a `TaskDataSet`, first construct a new `TaskDataSet`, then add inputs to it using the `addTaskInput` method. (See FIG. 63.) If one is using a stream, one can also use the `createTaskInput` method. After one has finished adding inputs, call the

doneSubmitting method. If a name is assigned using setName, that will be used for subsequent references to the session; otherwise, a name will be assigned. The set will remain on the Server until destroy is called, even if the Java VM that created it exits.

Creating a DataSetJob

- 5 After creating a TaskDataSet, implement the Job using DataSetJob, and TaskOutput. (See FIG. 64.) The main difference is that to run the Job, one must use setTaskDataSet to specify the dataset one created earlier. Note that the ExecuteLocally method cannot be used with the DataSetJob.

StreamJob and StreamTasklet

- 10 A StreamJob is a Job which allows one to create input and read output via streams rather than using defined objects. (See FIG. 65.) A StreamTasklet reads data from InputStream and writes to an OutputStream, instead of using a TaskInput or TaskOutput. When the StreamJob writes input to a stream, the data is written directly to the local file system, and given to Engines via a lightweight webserver. The Engine also streams the data in via the StreamTasklet. In this way, the memory overhead on the Driver, Broker, and Engine is reduced, since an entire TaskInput does not need to be loaded into memory for transfer or processing. The StreamTasklet must be used with StreamJob.

SQLDataSetJob and SQLTasklet

- 20 Engines can use information in an SQL database as input to complete a Task by means of SQL. An SQLDataSetJob queries the database and receives a result set. Each SQLTasklet is given a subset of the result set as an input. This feature is only available from the Java Driver.

Starting the database

- 25 To use an SQL database, one must first have a running database with a JDBC interface. (See FIG. 66.) The sample code loads a properties file called sqltest.properties which contains properties used by the database, plus the properties tasks and query, which are used in our Job. (See FIG. 67.)

SQLDataSetJob

An `SQLDataSetJob` is created by implementing `DataSetJob`. (See FIG. 67) Task inputs are not created, as they will be from the SQL database. (See FIG. 68.)

SQLTasklet

5 An `SQLTasklet` is implemented similar to a normal `Tasklet`, except the input is an SQL table. (See FIG. 69.)

Running the Job

After defining a `TaskOutput`, the Job can be run. The `SQLDataSet` is created on the server and is prepared with `setJDBCProperties`, `setMode`, `setQuery`, and `prepare`. Then the Job is run. (See FIG. 70.) Note that in order to use most recent
10 information in the database, the `SQLDataSet` needs to be destroyed and created again. This may be important if one is using a frequently updated database.

The Propagator API

This section discusses how to use the Propagator API to run parallel code with inter-node communication.

15 **Overview**

The Propagator API is a group of classes that can be used to distribute a problem over a variable number of compute Engines instead of fixed-node cluster. It is an appropriate alternative to MPI for running parallel codes which require inter-node communication. Unlike most MPI parallel codes, Propagator implementations can run over heterogeneous resources,
20 including interruptible desktop PCs.

A Propagator application is divided into steps, with steps sent to nodes. Using adaptive scheduling, the number of nodes can vary, even changing during a problem's computation. After a step has completed, a node can communicate with other nodes, propagating results and collecting information from nodes that have completed earlier steps. This checkpointing allows
25 for fault-tolerant computations.

FIG. 71 illustrates how nodes communicate at barrier synchronization points when each step of an algorithm is completed.

Using the Propagator API

The Propagator API consists of three classes: `GroupPropagator` and `NodePropagator`
30 and the Interface `GroupCommunicator`.

- The `GroupPropagator` is used as the controller. A `GroupPropagator` is created, and it is used to create the nodes and the messaging system used between nodes.
- The `NodePropagator` contains the actual code that each node will execute at each step. It also contains whatever code each node will need to send and receive messages, and send and receive the node state.
- The `GroupCommunicator` is the interface used by the nodes to send and receive messages, and to get and set node state.

Group Propagator

The `GroupPropagator` is the controlling class of the `NodePropagators` and `GroupCommunicator`. One should initially create a `GroupPropagator` as the first step in running a `Propagator Job`.

After creating a `GroupPropagator`, one can access the `GroupCommunicator`, like this:

```
GroupCommunicator gc = gp.getGroupCommunicator();
```

This will enable one to communicate with nodes, and get or set their state.

Next, one will need to set the `NodePropagator` used by the nodes. Given a simple `NodePropagator` implementation called `TestPropagator` that is passed the value of the integer `x`, one would do this:

```
gp.setNodePropagator( new TestPropagator( x ) );
```

After one has defined a `NodePropagator`, one can tell the nodes to execute a step of code by calling the `propagate` method, and passing a single integer containing the step number one wishes to run.

When a program is complete, the `endSession` method should be called to complete the session.

Node Propagator

The `NodePropagator` contains the actual code run on each node. The `NodePropagator` code is run on each step, and it communicates with the `GroupCommunicator` to send and receive messages, and set its state.

To create one's own `NodePropagator` implementation, create a class that extends `NodePropagator`. The one method the created class must implement is `propagate`. If

will be run when `propagate` is run in the `GroupPropagator`, and it contains the code which the node actually runs.

The code in the `NodePropagator` will vary depending on the problem. But several possibilities include getting the state of a node to populate variables with partial solutions, broadcasting a partial solution so that other nodes can use it, or sending messages to other nodes to relay work status or other information. All of this is done using the `GroupCommunicator`.

Group Communicator

The `GroupCommunicator` communicates messages and states between nodes and the `GroupPropagator`. It can also transfer the states of nodes. It's like the bus or conduit between all of the nodes.

The `GroupCommunicator` exists after one creates the `GroupPropagator`. It's passed to each `NodePropagator` through the `propagate` method. Several methods enable communication. They include the following (there are also variations available to delay methods until a specified step or to execute them immediately):

- `broadcast` Send a message to all recipients, except current node.
- `clearMessages` Clear all messages and states on server and Engines.
- `getMessages` Get the messages for current node.
- `getMessagesFromSender` Get the message from specified node for current node.
- `getNodeState` Get the state of specified node.
- `getNumNodes` Get the total number of nodes.
- `sendMessage` Send the message to `nodeId`.
- `setNodeState` Set the state of the node.

25 The 2-D Heat Equation - A Propagator API Example

We will now explain how to use the Propagator API to solve an actual problem. In this example, it is used to calculate a two-dimensional heat equation. This example uses three files: `Test.java`, which contains the main class, `HeatEqnSolver.java`, which implements the `GroupPropagator`, and `HeatPropagator`, which implements the `NodePropagator`.

30 Test.java

This file (see FIG. 72A) starts like most other LiveClusterprograms, except we import `com.livecluster.tasklet.propagator.*`. Also, a Test class is created as our main class.

Continuing (see FIG. 72B), properties are loaded from disk, and variables needed for the calculations are initialized, either from the properties file, or to a default value. If anything fails, an exception will be thrown.

Next (see FIG. 72C), the `GroupPropagator` is created. It's passed all of the variables it will need to do its calculations. Also, a message is printed to `System.out`, displaying the variables used to run the equation.

The solve method for the `HeatEqnSolver` object, which will run the equation, is called (see FIG. 72D), and the program ends.

HeatEqnSolver.java

The class `HeatEqnSolver` is defined with a constructor that is passed the values used to calculate the heat equation. It has a single public method, `Solve`, which is called by `Test` to run the program. (See FIG. 73A.) This creates the `GroupPropagator`, which controls the calculation on the nodes.

```
solver.solve();
```

A `GroupPropagator gp` is created (see FIG. 73B) with the name "heat2d," and the number of nodes specified in the properties. Then, a `GroupCommunicator gc` is assigned with the `GroupPropagator` method `getGroupCommunicator`. A new `HeatPropagator` is created, which is the code for the `NodePropagator`, which is described in the next section. The `HeatPropagator` is set as the `NodePropagator` for `gp`. It will now be used as the `NodePropagator`, and will have access to the `GroupCommunicator`. A Jarfile is set for the `GroupPropagator`.

The code (see FIG. 73C) then defines a matrix of random values and a mirror of the matrix for use by the nodes. After the math is done, the `i` loop uses `setNodeState` to push the value of the matrix to the nodes. Now, all of the nodes will be using the same starting condition for their calculations.

The main iteration loop (see below) uses the `propagate` method to send the steps to the nodes. This will cause `_iters` number of iterations by the nodes using their code.

```
// main iteration loop
for ( int i=0; i < _iters; i++ ) {
    gp.propagate(i);
}
```

5 As nodes return their results, the code (see **FIGs. 73D-E**) uses `getNodeState` to capture back the results and copy them into the matrix.

HeatPropagator.java

The `HeatPropagator` class (see **FIG. 74**) implements the `NodePropagator`, and is the code that will actually run on each node. When created, it is given `lastIter`, `fax` and
10 `facy`. It obtains the boundary information as a message from the last step that was completed. It completes its equations, then broadcasts the results so the next node that runs can continue.

The first thing `propagate` does is use `getNodeState` to initialize its own copy of the matrix. (See **FIG. 75A**.)

Next, boundary calculations are obtained. (See **FIG. 75B**.) These are results that are on
15 the boundary of what this node will calculate. If this is the first node, there aren't any boundaries, and nothing is done. But if this isn't step 0, there will be a message waiting from the last node, and it's obtained with `getMessagesFromSender`.

Next, the actual calculation takes place (see **FIG. 75C**), and then copied back into the matrix. The matrix is then set into the node state for the next iteration using `setNodeState`.
20 (see **FIG. 75D**.) The boundaries are also sent on for the next node using `sendMessage`.

This section explains how to use Engine Discriminators, a powerful method of controlling which Engines are eligible to receive specific Jobs.

About Discriminators

In a typical business environment, not every PC will be identical. Some departments
25 may have slower machines that are utilized less. Other groups may have faster PCs, but it may be a priority to use them to capacity during the day. And server farms of dedicated machines may be available all the time, without being interrupted by foreground tasks.

Depending on the Jobs one has and the general demographics of one's computing environment, the scheduling of Tasks to Engines may not be linear. And sometimes, a specific
30 Job may require special handling to ensure the optimal resources are available for it. Also, in

some LiveCluster installations, you one want to limit what Engines report to a given Broker for work. Or, one may want to limit what Driver submits work to a given Broker.

A **discriminator** enables one to specify what Engines can be assigned to a Task, what Drivers can submit Tasks to a Broker, and what Engines can report to a Broker. These
 5 limitations are set based on properties given to Engines or Drivers. Task discrimination is set in the Driver properties, and controls what Engines can be assigned to a Task. Broker discrimination is set in the LiveClusterAdministration Tool, and controls what Drivers and Engines use that Broker.

For example: say one is implementing LiveCluster at a site that has 1000 PCs.
 10 However, 300 of the PCs are slower machines used by the Marketing department, and they are rarely idle. The Job will require a large amount of CPU time from each Engine processing tasks. Without using discriminators, the Tasks are sent to the slower machines and are regularly interrupted. This means that roughly 30% of the time, a Task will be scheduled on a machine that might not complete any work.

15 Discriminators provide a solution to this issue. First, one would deploy Engines to all of one's computers; Marketing computers would have a department property set to Marketing, and the rest of the machines in the company would have the department property set to something other than Marketing. Next, when the application sends a complex Job with the LiveClusterAPI, it attaches a Task discriminator specifying not to send any Tasks from the Job
 20 to any Engine with the department property set to Marketing. The large Job's Tasks will only go to Engines outside of Marketing, and smaller Jobs with no Task discriminator set will have Tasks processed by any Engine in the company, including those in Marketing.

Configuring Engines with Properties

Default Properties

25 An Engine has several properties set by default, with values corresponding to the configuration of the PC running the Engine. One can use these properties to set discriminators. The default properties, available in all Engines, are as follows:

- **guid** The GUID (network card address)
- **id** The numerical ID of the Engine
- 30 • **instance** The instance, for multi-processor machines

- `username` The Engine's username
- `cpuNo` The number of CPUs on the machine
- `cpuMFlops` The performance, in Megaflops
- `totalMemInKB` Total available memory, in Kilobytes
- 5 • `freeMemInKB` Free memory, in Kilobytes
- `freeDiskInMB` Free disk space, in Megabytes
- `os` Operating system (win32, solaris or linux)

Custom Properties

To set other properties, one can add the properties to the Engine Tracker, and install
 10 the Engine using tracking. One may also add and changes properties individually after
 installation using the **Engine Properties** command.

In Windows:

To add custom properties to an Engine, in the LiveCluster Administration Tool, one
 must make changes using the **Engine Tracking Editor**. After one changes the properties in the
 15 editor, one will be prompted for values for the properties each time one installs an Engine with
 the **1-Click Install with Tracking** option. One can also change these at any time on any
 Engine with the **Engine Properties** command.

To access the editor, go to the **Configure** section, and click **Engine Tracking Editor**.

By default, the following properties are defined:

- 20 • `MachineName` hostname of the machine where the Engine is being
 installed;
- `Group` work group to attach Engine;
- `Location` machine location;
- `Description` brief description of machine.

25 When one installs an Engine with the **1-Click Install with Tracking** option, one will be
 prompted to enter values for all four of the properties. If one doesn't want to use all four
 properties, one may click the **Remove** button next to the properties one does not want to use.
 (Note that you cannot remove the `MachineName` property.)

To add another property to the above list, enter the property name in the **Property**
 30 column, then enter a description of the property in the **Description** column, and click **Add**.

Configuring Driver Properties

Broker discrimination can be configured to work on either Engines or Drivers. For discrimination on Drivers, one can add or modify properties in the `driver.properties` file included in the top-level directory of the Driver distribution.

5 Configuring Broker Discriminators

One can configure a Broker to discriminate which Engines and Drivers from which it will accept login sessions. This can be done from the LiveClusterAdministration Tool by selecting **Broker Discrimination** in the **Configure** section.

10 First, select the Broker to be configured from the list at the top of the page. If one is only running a single Broker, there will only be one entry in this list.

One can configure discriminators for both Driver properties and Engine properties. For Drivers, a discriminator is set in the Driver properties, and it prevents Tasks from a defined group of Drivers from being taken by this Broker. For Engines, a discriminator prevents the Engine from being able to log in to a Broker and take Tasks from it.

15 Each discriminator includes a property, a comparator, and a value. The property is the property defined in the Engine or Driver, such as a group, OS or CPU type. The value can be either a number (double) or string. The comparator compares the property and value. If they are true, the discriminator is matched, and the Engine can accept a Task, or the Driver can submit a Job. If they are false, the Driver is returned the Task, or in the case of an Engine, the
20 Broker will try to send the Task to another Engine.

The following comparators are available:

- **equals** A string that must equal the client's value for the property.
- **not equals** A string that must not equal the client's value for the property.
- **includes** A comma-delimited string that must equal the client's value for
25 that property. ("*" means accept all.)
- **excludes** A comma-delimited string that cannot equal the client's value for
that property. ("*" means deny all.)
- **=** The value is a number (double, for any to be used) that must equal
the value for that property.

- **!=** The value is a number (double, for any to be used) that must not equal the value for that property.
- **<** The value is a number, the client's value must be less than this value.
- 5 • **<=** The value is a number, the client's value must be less than or equal to this value.
- **>** The value is a number, the client's value must be greater than this value.
- **>=** The value is a number, the client's value must be greater than or
10 equal to this value.

One further option for each discriminator is the **Negate other Brokers** box. When this is selected, an Engine or Driver will be considered only for this Broker, and no others. For example, if one has a property named `state` and one sets a discriminator for when `state` equals `NY` and selects **Negate other Brokers**, any Engine with `state` set to `NY` will only go
15 to this Broker and not others.

Once you have entered a property, comparator, and value, click **Add**. One can add multiple discriminators to a Broker by defining another discriminator and clicking **Add** again. Click **Save** to save all added discriminators to the Broker.

By default, if an Engine or Driver does not contain the property specified in the
20 discriminator, the discriminator is not evaluated and considered false. However, one can select **Ignore Missing Properties** for both the Driver and Engine. This makes an Engine or Driver missing the property specified in a discriminator ignore the discriminator and continue. For example, if one sets a discriminator for `OS = Linux`, and an Engine doesn't have an `OS` property, normally the Broker won't give the Engine Jobs. But if one selects **Ignore Missing
25 Properties**, the Engine without properties will still get Jobs from the Broker.

Task discriminators are set by the Driver, either in Java or in XML. (See FIG. 76.)
The LiveClusterTutorial

This section provides details on how to obtain examples of using the LiveClusterAPI.
Using JNI Example

Often, the application, or some portion of it, is written in another (native) programming language such as C, C++, or Fortran, but it is convenient to use Java as the glue that binds the compute server to the application layer. In these cases the **Java Native Interface (JNI)** provides a simple mechanism for passing data and function calls between Java and the native code. [Note: One must create a separate wrapper to access the dynamically linked library (.dll or .so) from the Engine-side and insert a call to this wrapper in the service() method of the Tasklet interface.]

FIGs. 77-79 provide an example of a JNI for the previously-discussed Pi calculation program.

10 Submitting a LiveCluster Job

Using Java, jobs can be submitted to a LiveCluster Server in any of three ways:

- From the command line, using XML scripting:

```
java -cp DSDriver.jar MyApp picalc.xml
```

This method uses properties from the driver.properties file located in the same directory as the Driver. One can also specify command-line properties.

- At runtime using one of the createJob methods (this supports partial scripting of the Job Bean).

```
PiCalcJob job = (PiCalcJob) Job.createJob( new
File(picalc.xml) );
```

```
20 job.execute();
double pi = job.getPValue();
```

- At runtime (entirely).

```
PiCalcJob job = new PiCalcJob();
job.getOptions().setJarFile( new File(picalc.jar) );
25 job.setIterations(30000000);
job.setNumTasks(500);
job.execute();
double pi = job.getPValue();
```

XML scripting also supports the Batch object, which enables one to submit a Job once and have it run many times on a regular schedule.

Using C++, jobs must be submitted to a LiveClusterServer using the run-time interface:

```

    job = new PiJob();
    try {
        job->execute();// or executeInThread() or executeLocally()
    } catch (JobException je) {
5      cerr << "testJob caught an exception " << je << endl;
    }
    delete job;

```

Driver Properties

Properties can be defined in the `driver.properties` file, located in the same directory as the Driver. One can edit this file and add properties, as `property=value` pairs. One can also specify properties on the command line using the `-D` switch, if they are prefixed with `ds`. For example:

```

    java -Dds.DSPPrimaryDirector=server1:80 -
        Dds.DSSSecondaryDirector=server2:80
15    -cp DSDriver.jar MyApp picalc.xml

```

Properties specified on the command line are overwritten by properties specified in the `driver.properties` file. If one wants to set a property already defined in the `driver.properties`, one must first edit the `driver.properties` and comment out the property.

20 Using the Direct Data Transfer Property

Direct data transfer is enabled by setting `DSDirectDataTransfer=true`, which is the default setting in the `driver.properties` file. If one writes a shell script to create Jobs, each with their own Driver running from its own Java VM, one's script must provide a different port number for the `DSWebserverPort` property normally set in the

25 `driver.properties` file. If one's script instantiates multiple Drivers from the same `driver.properties` file with the same port number, the first Driver will open a web server listening to the defined socket. Subsequent Drivers will not open another web server as long as the first Job is running, but will be able to continue running by using the first Job's server for direct data. However when the first Job completes, its server will be terminated,

30 causing subsequent Jobs to fail.

To write a shell script for the above situation, one could remove the `DSWebserverPort` property from the `driver.properties` file and set a unique port number for each Job using a command line property, as described in the previous section.

XML Job Scripting

5 LiveCluster is packaged with XML-based scripting facilities one can use to create and configure Jobs. (see FIG. 80.) Since Java Jobs are JavaBeans components, their properties can be manipulated via XML and other Bean-compatible scripting facilities.

Batch Jobs

10 Jobs can be scheduled to run on a regular basis. Using XML scripting, one can submit a Job with specific scheduling instructions. Instead of immediately entering the queue, the Job will wait until the time and date specified in the instructions given.

Batch Jobs can be submitted to run at a specific absolute time, or a relative time, such as every hour. Also, a Batch Job can remain active, resubmitting a Job on a regular basis.

See, for example, FIG. 81, which submits the Linpack test at 11:20 AM on September 15 28th, 2001. The `batch` element contains the entire script, while the `schedule` element contains properties for `type` and `startTime`, defining when the Job will run. `job` actually runs the Job when it is time, and contains properties needed to run the Job, while `command` also runs at the same time, writing a message to a log.

Distributing Libraries, Shared Data, and Native Code

20 The LiveCluster system provides a simple, easy-to-use mechanism for distributing linked libraries (`.dll` or `.so`), Java class archives (`.jar`), or large data files that change relatively infrequently. The basic idea is to place the files to be distributed within a reserved directory associated with the Server. The system maintains a synchronized replica of the reserved directory structure for each Engine. This is called directory replication.

25 By default, four directories are replicated to Engines: `win32`, `solaris`, and `linux` directories are mirrored to Engines run on the respective operating systems, and `shared` is mirrored to all Engines.

The default location for these four directories are as follows:

30 `public_html/updates/resources/shared/`
 `public_html/updates/resources/win32/`

```
public_html/updates/resources/solaris/  
public_html/updates/resources/linux/
```

On the Server, these paths are relative to one's installation directory. For example, if one installs LiveCluster at c:\DataSynapse, one should append these paths to

- 5 C:\DataSynapse\Server\livecluster on your server. On the Engine, the default installation in Windows puts the shared and win32 directories in C:\Program Files\DataSynapse\Engine\resources.

To configure directory replication, in the Administration Tool, go to the **Configure** section, and select **Broker Configuration**. Select **Engine Manager**, then **Engine File Update**
10 **Server**.

When **Auto Update Enabled** is set to true (the default), the shared directories will automatically be mirrored to any Engine upon login to the Broker. Also, the Server will check for file changes in these directories at the time interval specified in **Minutes Per Check**. If changes are found, all Engines are signaled to make an update.

- 15 One can force all Engines to update immediately by setting **Update All Now** to true. This will cause all Engines to update, and then its value will return to false. If one has installed new files and wants all Engines to use them immediately, set this option to true.

Verifying the Application

- Before deploying any application in a distributed environment, one should verify that it
20 operates correctly in a purely local setting, on a single processor. The `executeLocally()` method in the `Job` class is provided for this purpose. Calling this method results in synchronous execution on the local processor; that is, the constituent Tasks execute sequentially on the local processor, without any intermediation from a Broker or distribution to remote Engines.

25 Optimizing LiveCluster Server architecture

The LiveCluster Server architecture can be deployed to give varying degrees of redundancy and load sharing, depending on the computing resources available. Before installation, it's important to ascertain how LiveCluster will be used, estimate the volume and frequency of jobs, and survey what hardware and networking will be used for the installation.

First, it's important to briefly review the architecture of a Server. The LiveCluster Server consists of two entities: the LiveCluster Director and the LiveCluster Broker:

- **Director** — Responsible for authenticating Engines and initiating sessions between Engines and Brokers, or Drivers and Brokers. Each LiveCluster installation must have a **Primary Director**. Optionally, a LiveCluster installation can have a **Secondary Director**, to which Engines will log in if the Primary Director fails.
- **Broker** — Responsible for managing jobs by assigning tasks to Engines. Every LiveCluster installation must have at least one Broker, often located on the same system as the primary Director. If more than one Broker is installed, then a Broker may be designated as a **Failover Broker**; it accepts Engines and Drivers only if all other Brokers fail.

A minimal configuration of LiveCluster would consist of a single Server configured as a Primary Director, with a single Broker. Additional Servers containing more Brokers or Directors can be added to address three primary concerns: redundancy, volume, and other considerations.

Redundancy

Given a minimal configuration of a single Director and single Broker, Engines and Drivers will log in to the Director, but failure of the Director (either by excessive volume, Server failure, or network failure) would mean a Driver or Engine not logged in would no longer be able to contact a Director to establish a connection.

To prevent this, redundancy can be built into the LiveCluster architecture. One method is to run a second Server with a Secondary Director, and configure Engines and Drivers with the address of both Directors. When the Primary Director fails, the Engine or Driver will contact the Secondary Director, which contains identical Engine configuration information and will route Engines and Drivers to Brokers in the same manner as the Primary Director. FIG. 82 shows an exemplary implementation with two Servers.

In addition to redundant Directors, a Broker can also have a backup on a second Server. A Broker can be designated a Failover Broker on a second Server during installation. Directors will only route Drivers and Engines to Failover Brokers if no other regular Brokers are available. When regular Brokers then become available, nothing further is routed to the

Failover Broker. When a Failover Broker has finished processing any remaining jobs, it logs off all Engines, and Engines are then no longer routed to that Failover Broker. FIG. 82 shows a Failover Broker on the second Server.

Volume

5 In larger clusters, the volume of Engines in the cluster may require more capability than can be offered by a single Broker. To distribute load, additional Brokers can be added to other Servers at installation. For example, FIG. 83 shows a two Server system with two Brokers. Drivers and Engines will be routed to these Brokers in round-robin fashion.

Other Considerations

10 Several other factors may influence how one may integrate LiveCluster with an existing computing environment. These include:

- Instead of using one Cluster for all types of Jobs, one may wish to segregate different subsets of jobs (for example, by size or priority) to different Directors.
- One's network may dictate how the Server environment should be planned. For
15 example, if one has offices in two parts of the country and a relatively slow extranet but a fast intranet in each location, one could install a Server in each location.
- Different Servers can support data used for different job types. For example, one Server can be used for Jobs accessing a SQL database, and a different Server can be used for jobs that don't access the database.

20 With this flexibility, it's possible to architect a Server model to provide a job space that will facilitate job traffic.

Configuring a network

 Since LiveCluster is a distributed computing application, successful deployment will depend on one's network configuration. LiveCluster has many configuration options to help it
25 work with existing networks. LiveCluster Servers should be treated the same way one treats other mission-critical file and application servers: assign LiveCluster Servers static IP addresses and resolvable DNS hostnames. LiveCluster Engines and Drivers can be configured in several different ways. To receive the full benefit of peer-to-peer communication, one will need to enable communication between Engines and Drivers (the default), but LiveCluster can also be
30 configured to work with a hub and spoke architecture by disabling Direct Data Transfer.

Name Service

LiveCluster Servers should run on systems with static IP addresses and resolvable DNS hostnames. In a pure Windows environment, it is possible to run LiveCluster using just WINS name resolution, but this mode is not recommended for larger deployments or heterogeneous environments.

Protocols and Port Numbers

LiveCluster uses the Internet Protocol (IP). All Engine-Server, Driver-Server, and Engine-Driver communication is via the HTTP protocol. Server components, Engines, and Drivers can be configured to use port 80 or any other available TCP port that is convenient for one's network configuration.

All Director-Broker communication is via TCP. The default Broker login TCP port is 2000, but another port can be specified at installation time. By default, after the Broker logs in, another pair of ephemeral ports is assigned for further communication. The Broker and Director can also be configured to use static ports for post-login communication.

Server-Engine and Driver-Server Communication

All communication between Engines and Servers (Directors and Brokers) and between Drivers and Servers is via the HTTP protocol, with the Engine or Driver acting as HTTP client and the Server acting as HTTP server. (See FIG. 84.)

The Server can be configured to work with an NAT device between the Server and the Engines or Drivers. To do this, specify the external (translated) address of the NAT device when referring to the Server address in Driver and Engine installation.

Win32 LiveCluster Engines can also support an HTTP proxy for communication between the Engine and the Broker. If the default HTML browser is configured with an HTTP proxy, the Win32 Engine will detect the proxy configuration and use it. However, since all LiveCluster communication is dynamic, the HTTP proxy is effectively useless, and for this reason it is preferred not to use an HTTP proxy.

Broker-Director Communication

Communication between Brokers and Directors is via TCP. (See FIG. 85.) By default, the Broker will log in on port 2000, and ephemeral ports will then be assigned for further communication. This configuration does not permit a firewall or screening router between the

Brokers and Directors. If a firewall or screening router must be supported between Brokers and Directors, then the firewall or screening must have the Broker login port (default 2000) open. Additionally, the Brokers must be configured to use static ports for post-login communication, and those ports must be open on the firewall as well.

5 Direct Data Transfer

By default, LiveCluster uses Direct Data Transfer, or peer-to-peer communication, to optimize data throughput between Drivers and Engines. (See FIGs. 86-87.) Without Direct Data Transfer, all task inputs and outputs must be sent through the Server. Sending the inputs and outputs through the Server will result in higher memory and disk use on the Server, and
10 lower throughput overall.

With Direct Data Transfer, only lightweight messages are sent through the Server, and the "heavy lifting" is done by the Driver and Engine nodes themselves. Direct data transfer requires that each peer knows the IP address that he presents to other peers. In most cases, therefore, Direct Data Transfer precludes the use of NAT between the peers. Likewise, Direct
15 Data Transfer does not support proxies.

For LiveCluster deployments where NAT is already in effect, NAT between Drivers and Engines can be supported by disabling peer-to-peer communication as follows:

- If, from the perspective of the Drivers, the Engines appear to be behind an NAT device, then the Engines cannot provide peer-to-peer communication, because they
20 won't know their NAT address. In this case Direct Data Transfer must be disabled in the Engine configuration.
- Likewise, if, from the perspective of the Engines, the Drivers appear to be behind an NAT device, then the Drivers cannot provide peer-to-peer communication, as they do not know their NAT address. In this case Direct Data Transfer must be disabled in the
25 Driver properties.

While the foregoing has described the invention by recitation of its various aspects/features and illustrative embodiment (s) thereof, those skilled in the art will recognize that alternative elements and techniques, and/or combinations and sub-combinations of the described elements and techniques, can be substituted for, or added to, those described herein.
30 The present invention, therefore, should not be limited to, or defined by, the specific

apparatus, methods, and articles-of-manufacture described herein, but rather by the appended claims (and others that may be contained in continuing applications), which claims are intended to be construed in accordance with well-settled principles of claim construction, including, but not limited to, the following :

- 5 • Limitations should not be read from the specification or drawings into the claims (i.e., if the claim calls for a "chair," and the specification and drawings show a rocking chair, the claim term "chair" should not be limited to a rocking chair, but rather should be construed to cover any type of "chair").
- 10 • The words "comprising," "including," and "having" are always open-ended, irrespective of whether they appear as the primary transitional phrase of a claim, or as a transitional phrase within an element or sub-element of the claim (e.g., the claim "a widget comprising: A; B; and C" would be infringed by a device containing 2A's, B, and 3C's; also, the claim "a gizmo comprising: A; B, including X, Y, and Z; and C, having P and Q" would be infringed by a device containing 3A's, 2X's, 3Y's, Z, 6P's, and Q).
- 15 • The indefinite articles "a" or "an" mean "one or more"; where, instead, a purely singular meaning is intended, a phrase such as "one," "only one," or "a single," will appear.
- 20 • Where the phrase "means for" precedes a data processing or manipulation "function," it is intended that the resulting means-plus-function element be construed to cover any, and all, computer implementation(s) of the recited "function" using any standard programming techniques known by, or available to, persons skilled in the computer programming arts.

- A claim that contains more than one computer-implemented means-plus-function element should not be construed to require that each means-plus-function element must be a structurally distinct entity (such as a particular piece of hardware or block of code); rather, such claim should be construed merely to require that the overall
5 combination of hardware/firmware/software which implements the invention must, as a whole, implement at least the function(s) called for by the claim.

In light of the above, and reserving all rights to seek additional claims covering the subject matter disclose above,

WHAT WE CLAIM IN THIS APPLICATION IS:

1. A distributed computing system, comprising:
5 a plurality of engines;
at least one broker;
at least on client application, said client application having an associated driver;
said driver configured to enable communication between said client application and two
or more of said engines via a peer-to-peer communication network;
10 characterized in that said driver is further configured to enable communication between
said client application and said at least one broker over said peer-to-peer network, and said
broker is further configured to communicate with said engines over said peer-to-peer network,
thereby enabling said broker to control and supervise the execution of tasks provided by said
client application on said two or more engines.
- 15 2. A distributed computing system, as defined in claim 1, further including at least
one failover broker configured to communicate with said driver and said engines, and, in the
event of a broker failure, control and supervise the execution of tasks provided by said client
application on said two or more engines.
- 20 3. A distributed computing system, as defined in claim 1, wherein said broker
further includes an adaptive scheduler configured to selectively assign and control the
execution of tasks provided by said client application on said engines.
- 25 4. A distributed computing system, as defined in claim 3, wherein said adaptive
scheduler is further configured to redundantly assign one or more of the task(s) provided by
said client application to multiple engines, so as to ensure the timely completion of said
redundantly assigned task(s) by at least one of said engines.
5. A distributed computing system, as defined in claim 1, wherein the tasks
provided by said client application have associated discriminators.
- 30 6. A distributed computing system, as defined in claim 5, wherein said broker
utilizes parameters associated with said discriminators and said engines to determine the
assignment of tasks to engines.

7. A distributed computing system, as defined in claim 1, wherein said system controls the timing of selected communications between said driver and said engines so as to avoid bottlenecks associated with overloads of said peer-to-peer network.

8. A distributed computing system, as defined in claim 1, wherein said system is
5 further configured to selectively delay certain communications over said peer-to-peer network, thereby avoiding excessive simultaneous network traffic and increasing overall performance of said system.

9. A distributed computing system, as defined in claim 1, wherein said broker and said two or more engines each include an associated propagator object that permits control
10 over engine-to-engine propagation of data over said peer-to-peer network.

10. A distributed computing system, as defined in claim 9, wherein said propagator objects enable an engine or broker node to perform at least three of the following operations:

- (i) broadcast a message to all nodes, except the current node;
- (ii) clear all message(s), and associated message state(s), on specified broker(s)
15 and/or engine(s);
- (iii) get message(s) for the current node;
- (iv) get the message(s) from a specified node for the current node;
- (v) get the state of a specified node;
- (vi) get the total number of nodes;
- 20 (vii) send a message to a specified node; and/or,
- (viii) set the state of a specified node.

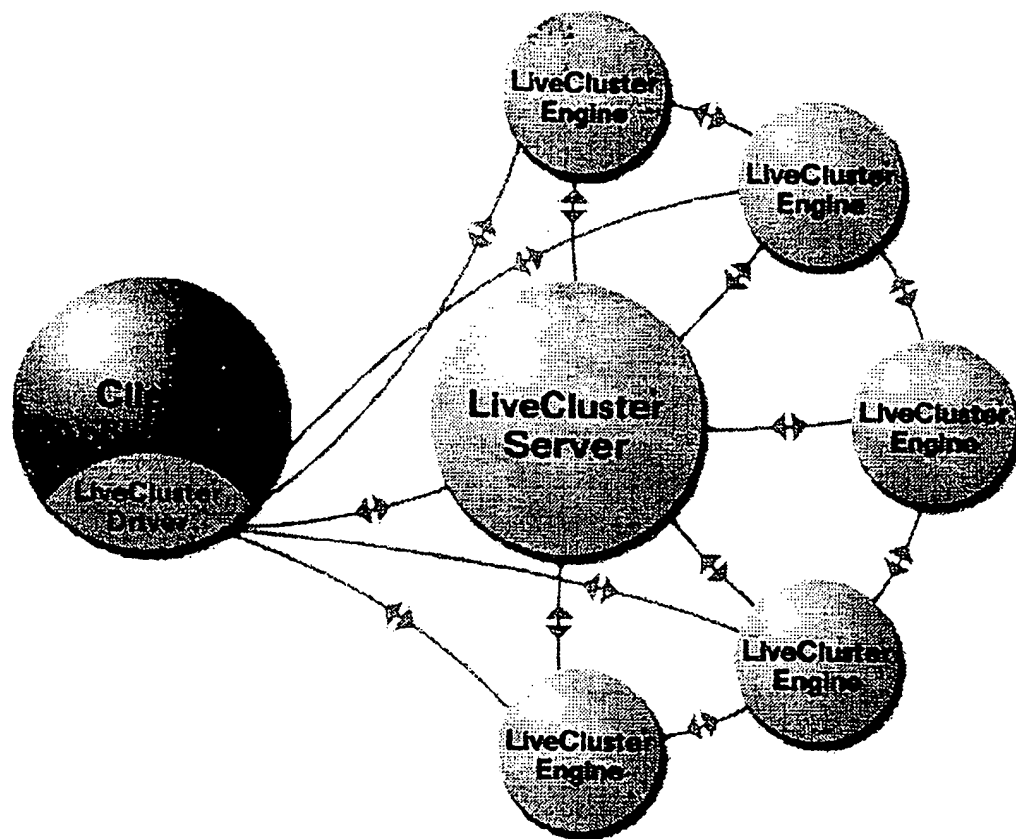
11. A distributed computing system, as defined in claim 9, wherein said propagator objects enable an engine or broker node to perform at least five of the following operations:

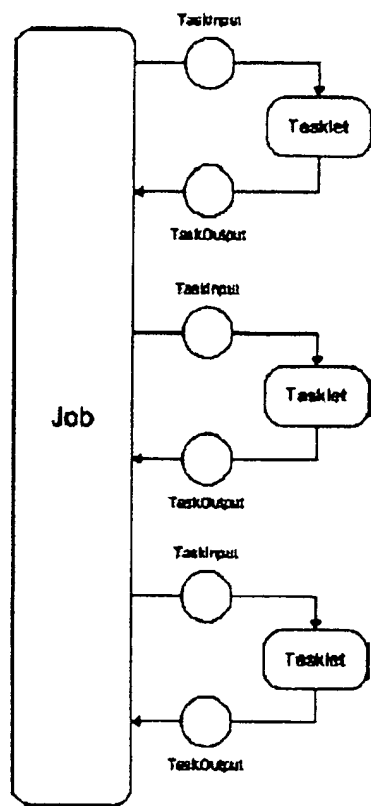
- (i) broadcast a message to all nodes, except the current node;
- 25 (ii) clear all message(s), and associated message state(s), on specified broker(s) and/or engine(s);
- (iii) get message(s) for the current node;

- (iv) get the message(s) from a specified node for the current node;
- (v) get the state of a specified node;
- (vi) get the total number of nodes;
- (vii) send a message to a specified node; and/or,
- 5 (viii) set the state of a specified node.

12. A distributed computing system, as defined in claim 9, wherein said propagator objects enable an engine or broker node to perform at least five of the following operations:

- (i) broadcast a message to all nodes, except the current node;
- (ii) clear all message(s), and associated message state(s), on specified broker(s)
- 10 and/or engine(s);
- (iii) get message(s) for the current node;
- (iv) get the message(s) from a specified node for the current node;
- (v) get the state of a specified node;
- (vi) get the total number of nodes;
- 15 (vii) send a message to a specified node; and/or,
- (viii) set the state of a specified node.

**FIG. 1**

**FIG. 2**

```
package tutorial.ch1_hello;

import com.livecluster.tasklet.*;

class SimpleTaskInput implements TaskInput {
    int taskId;
}
```

FIG. 3

```
package tutorial.ch1_hello;  
  
import com.livecluster.tasklet.*;  
  
class SimpleTaskOutput implements TaskOutput {  
    String s;  
}
```

FIG. 4

```
package tutorial.ch1_hello;

import com.livecluster.tasklet.*;

class HelloTasklet implements Tasklet {
    public TaskOutput service(TaskInput taskInput) {
        int myId = ((SimpleTaskInput) taskInput).taskId;
        SimpleTaskOutput output = new SimpleTaskOutput();
        output.s = "Hello from #" + myId;
        return output;
    }
}
```

FIG. 5

```
package tutorial.ch1_Hello;

import com.livecluster.tasklet.*;

class HelloStandaloneTest {
    public static void main(String[] args) {
        HelloTasklet tasklet = new HelloTasklet();
        for (int i = 0; i < 10; i++) {
            SimpleTaskInput input = new SimpleTaskInput();
            input.taskId = i;
            TaskOutput output = tasklet.service(input);
            System.out.println(((SimpleTaskOutput) output).s);
        }
    }
}
```

FIG. 6

```
package tutorial.ch1_hello;

import com.livecluster.tasklet.*;

public class HelloJob extends Job {
    public HelloJob() {
        setTasklet(new HelloTasklet());
    }

    // Pass each task a unique number to identify it.
    public void createTaskInputs() throws Exception {
        for (int i = 0; i < 10; i++) {
            SimpleTaskInput input = new SimpleTaskInput();
            input.taskId = i;
            addTaskInput(input);
        }
    }

    // Display the output of each task.
    public void processTaskOutput(TaskOutput output) {
        SimpleTaskOutput sto = (SimpleTaskOutput) output;
        System.out.println(sto.s);
    }
}
```

FIG. 7

```
package tutorial.ch1_hello;

import com.livecluster.tasklet.*;

class Test {
    public static void main(String[] args) throws Exception {
        Job job = new HelloJob();
        job.getOptions().setJarFile(new
            java.io.File("./jars/ch1_hello.jar"));
        job.execute();
        System.out.println("DONE");
    }
}
```

FIG. 8

```
public abstract class Deal {  
    public Deal(int id) { _id = id; }  
    public abstract Valuation value(PricingEnvironment env);  
  
    // other methods omitted  
  
    protected int _id;  
}
```

FIG. 9


```
public class ZeroCouponBond extends Deal {
    public ZeroCouponBond(int id, Date maturity,
        double principal) {
        super(id);
        _maturity = maturity;
        _principal = principal;
    }

    public Valuation value(PricingEnvironment env) {
        double years = DateUtil.yearsBetween(
            env.getValuationDate(), _maturity);
        double rate = env.getInterestRate(years);
        double val = _principal * Math.exp(-rate*years);
        return new Valuation(_id, val);
    }

    private Date _maturity;
    private double _principal;
}
```

FIG. 10

```
public Deal getDeal(Integer dealId) {  
    Deal t = (Deal) dealMap.get(dealId);  
    if (t == null) {  
        int maturityYears = (int) (Math.random()*10) + 1;  
        double principal = Math.random()*10000;  
        Calendar c = Calendar.getInstance();  
        c.add(Calendar.YEAR, maturityYears);  
        t = new ZeroCouponBond(dealId.intValue(),  
                                c.getTime(), principal);  
        dealMap.put(dealId, t);  
    }  
    return t;  
}
```

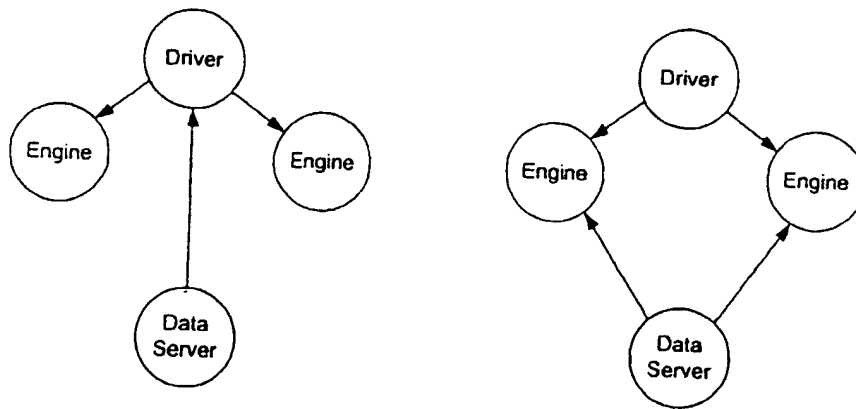
FIG. 11

```
package tutorial.ch2_valuation;

import java.util.*;

public class ValuationApp {
    public static void main(String[] args) {
        int totalDeals = 10;
        Date today = new Date();
        PricingEnvironment pe = new PricingEnvironment(today);
        for (int i = 0; i < totalDeals; i++) {
            Deal d = DealProvider.getInstance().
                getDeal(new Integer(i));
            Valuation v = d.value(pe);
            System.out.println(v);
        }
    }
}
```

FIG. 12

**FIG. 13**

```
public class ArrayListTaskIO implements TaskInput, TaskOutput {  
    private ArrayList list;  
    public ArrayListTaskIO() {  
        list = new ArrayList();  
    }  
    // other methods omitted  
}
```

FIG. 14

```
class ValuationTasklet implements Tasklet {
    ValuationTasklet(PricingEnvironment pe) {
        _pricingEnvironment = pe;
    }

    public TaskOutput service(TaskInput input) {
        ArrayListTaskIO dealIds = (ArrayListTaskIO) input;
        ArrayListTaskIO output = new ArrayListTaskIO();
        for (int i = 0; i < dealIds.size(); i++) {
            Integer dealId = (Integer) dealIds.get(i);
            Deal deal =
DealProvider.getInstance().getDeal(dealId);
            output.add(deal.value(_pricingEnvironment));
        }
        return output;
    }
    private PricingEnvironment _pricingEnvironment;
}
```

FIG. 15

```
public void createTaskInputs() throws Exception {
    int dealsAdded = 0;

    while (dealsAdded < _totalDeals) {
        ArrayListTaskIO input = new ArrayListTaskIO();
        for (int i = 0; i < _dealsPerTask && dealsAdded <
            _totalDeals; i++)
            input.add(new Integer(dealsAdded++));
        addTaskInput(input);
    }
}
```

FIG. 16

```
protected void processTaskOutput(TaskOutput out) {  
    ArrayList vals = ((ArrayListTaskIO) out).getArrayList();  
    _valuations.addAll(vals);  
}
```

FIG. 17


```
public static void main(String[] args) throws Exception {
    Properties props = new Properties();
    props.load(new
FileInputStream("./properties/valuation.properties"));
    int totalDeals =
Integer.parseInt(props.getProperty("totalDeals"));
    int dealsPerTask =
Integer.parseInt(props.getProperty("dealsPerTask"));
    _job = new ValuationJob(totalDeals, dealsPerTask);
    _job.getOptions().setJarFile(new
File("./jars/ch2_valuation.jar"));
    PricingEnvironment pe = new PricingEnvironment();
    _job.setTasklet(new ValuationTasklet(pe));
    _job.execute();
    System.out.println(_job.getValuations());
}
```

FIG. 18

```
public TaskOutput service(TaskInput input) {
    try {
        Properties p = EngineSession.getProperties();
        System.out.println("Engine properties:");
        p.list(System.out);
        System.out.println();
        return null; // no TaskOutput needed
    } catch (Exception e) {
        throw new RuntimeException(e.toString());
    }
}
```

FIG. 19

```
class EnginePropertiesJob extends Job {  
  private static TaskInput ti = new TaskInput() {};  
  public void createTaskInputs() throws Exception {  
    for (int i = 0; i < 5; i++)  
      addTaskInput(ti);  
  }  
  // other methods omitted  
}
```

FIG. 20

Engine properties:
-- listing properties --
id=1007399349412
os=win32
username=mingus
freeDiskInMB=26433
cpuMFlops=90.2
cpuNo=1
lastUpdated=Fri Dec 07 11:40:58 EST 2001
instance=0
freeMemInKB=150300
totalMemInKB=523744

FIG. 21

```
public Valuation value(PricingEnvironment env) {  
    double v = nativeValue(env);  
    return new Valuation(_id, v);  
}
```

FIG. 22

```
public void createTaskInputs() throws Exception {
    Properties props = new Properties();
    props.setProperty("os.equals", "win32");
    PropertyDiscriminator discriminator = new
PropertyDiscriminator(props);
    createDealInputs(_totalZeroDeals, 0, null);
    createDealInputs(_totalOptionDeals,
DealProvider.MIN_OPTION_ID, discriminator);
}
```

FIG. 23

```
private void createDealInputs(int totalDeals, int startingDealId,
    IDiscriminator discriminator) throws Exception {
    int dealsAdded = 0;
    while (dealsAdded < totalDeals) {
        ArrayListTaskIO input = new ArrayListTaskIO();
        for (int i = 0; i < _dealsPerTask && dealsAdded <
totalDeals; i++)
            input.add(new Integer(startingDealId +
dealsAdded++));
        addTaskInput(input, discriminator);
    }
}
```

FIG. 24

```
class SearchTasklet extends StreamTasklet {  
    SearchTasklet(String target) {  
        _target = target;  
    }  
    //...
```

FIG. 25


```
public void service(InputStream input, OutputStream output) {
    try {
        BufferedReader in = new BufferedReader(new
InputStreamReader(input));
        PrintWriter out = new PrintWriter(output);
        try {
            String line;
            while ((line = in.readLine()) != null)
                if (line.indexOf(_target) >= 0)
                    out.println(line);
        } finally {
            in.close();
            out.close();
        }
    } catch (IOException e) {
        throw new RuntimeException(e.toString());
    }
}
```

FIG. 26

```
protected void createTaskInputs() throws Exception {
    // Split the data file into chunks of _linesPerTask lines.
    BufferedReader in = new BufferedReader(new
FileReader(_file));
    try {
        String line = in.readLine();
        while (line != null) {
            OutputStream os = createTaskInput();
            PrintWriter out = new PrintWriter(os);
            int lineCount = 0;
            do {
                out.println(line);
                lineCount++;
                line = in.readLine();
            } while (lineCount < _linesPerTask && line !=
null);
            out.close();
        }
    } finally {
        in.close();
    }
}
```

FIG. 27

```
protected void processTaskOutput(InputStream instream) {  
    // Input stream contains matching lines -- print them.  
    BufferedReader in = new BufferedReader(new  
    InputStreamReader(instream));  
    try {  
        String line;  
        while ((line = in.readLine()) != null)  
            System.out.println(line);  
    } catch (IOException e) {  
        e.printStackTrace();  
    } finally {  
        try { in.close(); } catch (IOException e) {  
e.printStackTrace(); }  
    }  
}
```

FIG. 28

```
class SearchJob extends DataSetJob {  
    SearchJob(String target, TaskDataSet dataSet) {  
        setTasklet(new SearchTasklet(target));  
        setTaskDataSet(dataSet);  
    }  
    // other methods omitted  
}
```

FIG. 29

```
public static void main(String[] args) throws Exception {
    Properties props =
    readProperties("./properties/datasetsearch.properties");
    File dataFile = new File(props.getProperty("dataFile"));
    int linesPerTask =
    Integer.parseInt(props.getProperty("linesPerTask"));
    String[] targets =
    parseCommaList(props.getProperty("targets"));
    TaskDataSet dataSet =
        createDataSetFromFile("string-search", dataFile,
    linesPerTask);
    for (int i = 0; i < targets.length; i++) {
        _job = new SearchJob(targets[i], dataSet);
        _job.getOptions().setJarFile(new
    File("./jars/ch6_datasetsearch.jar"));
        _job.execute();
        System.out.println("DONE WITH JOB #" + i);
    }
    System.out.println("ALL DONE");
}
```

FIG. 30

```
static TaskDataSet createDataSetFromFile(String name, File file,
    int linesPerTask) throws Exception {
    TaskDataSet dataSet = new TaskDataSet(name);
    BufferedReader in = new BufferedReader(new
FileReader(file));
    try {
        String line = in.readLine();
        while (line != null) {
            PrintWriter out = new
PrintWriter(dataSet.createTaskInput());
            int lineCount = 0;
            do {
                out.println(line);
                lineCount++;
                line = in.readLine();
            } while (lineCount < linesPerTask && line !=
null);
            out.close();
        }
        dataSet.doneSubmitting();
    } finally {
        in.close();
    }
    return dataSet;
}
```

FIG. 31



FIG. 32

#	Full Name	UserName	Email	Access
1	Samuel Rodriguez	Admin		Full

FIG. 33

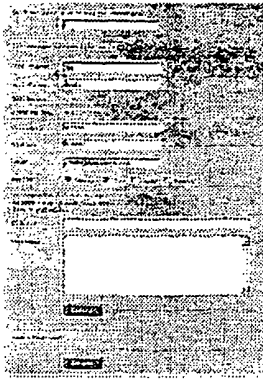
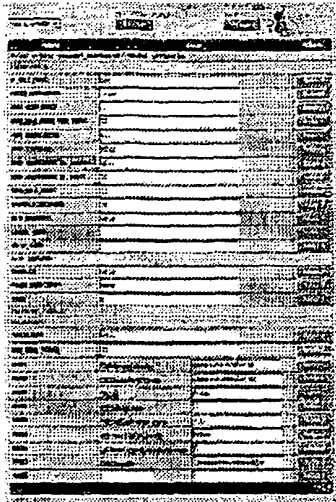


FIG. 34



The image shows a document page that is severely degraded. It appears to be a table or a form with multiple columns and rows. The text is almost entirely illegible due to extreme noise and poor quality. Some faint text is visible at the top and bottom of the page, but the main content is lost.

FIG. 35

Legend: RA (15442) / RA (15442) Name: Administrator

None: All: None: None: Change

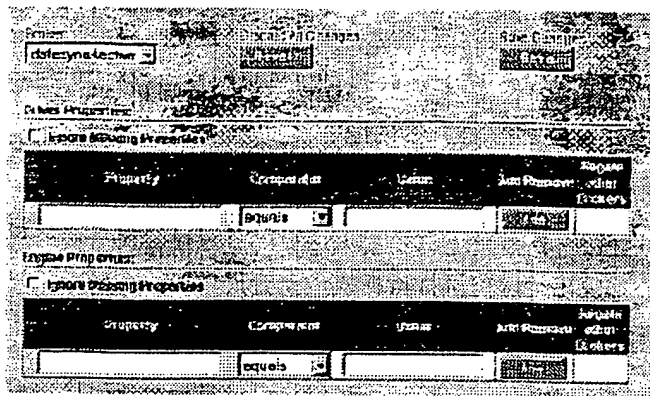
None: All: None: None: Change

Factory	State	Assembly	Inventory
Production	On	Production	Inventory
Test	On	Test	Inventory

FIG. 37

NAME		VALUE	SERIAL
1	1000	1000	1000
2	1000	1000	1000
3	1000	1000	1000
4	1000	1000	1000
5	1000	1000	1000
6	1000	1000	1000
7	1000	1000	1000
8	1000	1000	1000
9	1000	1000	1000
10	1000	1000	1000
11	1000	1000	1000
12	1000	1000	1000
13	1000	1000	1000
14	1000	1000	1000
15	1000	1000	1000
16	1000	1000	1000
17	1000	1000	1000
18	1000	1000	1000
19	1000	1000	1000
20	1000	1000	1000
21	1000	1000	1000
22	1000	1000	1000
23	1000	1000	1000
24	1000	1000	1000
25	1000	1000	1000
26	1000	1000	1000
27	1000	1000	1000
28	1000	1000	1000
29	1000	1000	1000
30	1000	1000	1000
31	1000	1000	1000
32	1000	1000	1000
33	1000	1000	1000
34	1000	1000	1000
35	1000	1000	1000
36	1000	1000	1000
37	1000	1000	1000
38	1000	1000	1000
39	1000	1000	1000
40	1000	1000	1000
41	1000	1000	1000
42	1000	1000	1000
43	1000	1000	1000
44	1000	1000	1000
45	1000	1000	1000
46	1000	1000	1000
47	1000	1000	1000
48	1000	1000	1000
49	1000	1000	1000
50	1000	1000	1000
51	1000	1000	1000
52	1000	1000	1000
53	1000	1000	1000
54	1000	1000	1000
55	1000	1000	1000
56	1000	1000	1000
57	1000	1000	1000
58	1000	1000	1000
59	1000	1000	1000
60	1000	1000	1000
61	1000	1000	1000
62	1000	1000	1000
63	1000	1000	1000
64	1000	1000	1000
65	1000	1000	1000
66	1000	1000	1000
67	1000	1000	1000
68	1000	1000	1000
69	1000	1000	1000
70	1000	1000	1000
71	1000	1000	1000
72	1000	1000	1000
73	1000	1000	1000
74	1000	1000	1000
75	1000	1000	1000
76	1000	1000	1000
77	1000	1000	1000
78	1000	1000	1000
79	1000	1000	1000
80	1000	1000	1000
81	1000	1000	1000
82	1000	1000	1000
83	1000	1000	1000
84	1000	1000	1000
85	1000	1000	1000
86	1000	1000	1000
87	1000	1000	1000
88	1000	1000	1000
89	1000	1000	1000
90	1000	1000	1000
91			

FIG. 38

**FIG. 39**

Choose which reports and charts to view

Print Manager: ☐ Admin. ☐ Info

Tables	Charts
<input type="checkbox"/> Clean Indicators by State	<input type="checkbox"/> Map of the New or Pending (Approved) Indicators
<input type="checkbox"/> Clean Indicators by State	<input type="checkbox"/> Map of the New or Pending (Covered) Indicators
<input type="checkbox"/> Clean Indicators by State	<input type="checkbox"/> Map of the New or Pending (Other) Indicators
<input type="checkbox"/> Clean Indicators by State	<input type="checkbox"/> Map of the New or Pending (Other) Indicators

Navigation:

<input type="checkbox"/> Home	<input type="checkbox"/> About Us	<input type="checkbox"/> Help
<input type="checkbox"/> Home	<input type="checkbox"/> About Us	<input type="checkbox"/> Help
<input type="checkbox"/> Home	<input type="checkbox"/> About Us	<input type="checkbox"/> Help

Buttons:

Print Manager: ☐ Admin. ☐ Info

Buttons:

Print Manager: ☐ Admin. ☐ Info

FIG. 40

[illegible]

FIG. 41

TABLE 1						
Step	Input	Output	Input	Output	Input	Output
1	0	0	0	1	1	0
2	1	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
11	0	0	0	0	0	0
12	0	0	0	0	0	0
13	0	0	0	0	0	0
14	0	0	0	0	0	0
15	0	0	0	0	0	0
16	0	0	0	0	0	0
17	0	0	0	0	0	0
18	0	0	0	0	0	0
19	0	0	0	0	0	0
20	0	0	0	0	0	0
21	0	0	0	0	0	0
22	0	0	0	0	0	0
23	0	0	0	0	0	0
24	0	0	0	0	0	0
25	0	0	0	0	0	0
26	0	0	0	0	0	0
27	0	0	0	0	0	0
28	0	0	0	0	0	0
29	0	0	0	0	0	0
30	0	0	0	0	0	0
31	0	0	0	0	0	0
32	0	0	0	0	0	0
33	0	0	0	0	0	0
34	0	0	0	0	0	0
35	0	0	0	0	0	0
36	0	0	0	0	0	0
37	0	0	0	0	0	0
38	0	0	0	0	0	0
39	0	0	0	0	0	0
40	0	0	0	0	0	0
41	0	0	0	0	0	0
42	0	0	0	0	0	0
43	0	0	0	0	0	0
44	0	0	0	0	0	0
45	0	0	0	0	0	0
46	0	0	0	0	0	0
47	0	0	0	0	0	0
48	0	0	0	0	0	0
49	0	0	0	0	0	0
50	0	0	0	0	0	0
51	0	0	0	0	0	0
52	0	0	0	0	0	0
53	0	0	0	0	0	0
54	0	0	0	0	0	0
55	0	0	0	0	0	0
56	0	0	0	0	0	0
57	0	0	0	0	0	0
58	0	0	0	0	0	0
59	0	0	0	0	0	0
60	0	0	0	0	0	0
61	0	0	0	0	0	0
62	0	0	0	0	0	0
63	0	0	0	0	0	0
64	0	0	0	0	0	0
65	0	0	0	0	0	0
66	0	0	0	0	0	0
67	0	0	0	0	0	0
68	0	0	0	0	0	0
69	0	0	0	0	0	0
70	0	0	0	0	0	0
71	0	0	0	0	0	0
72	0	0	0	0	0	0
73	0	0	0	0	0	0
74	0	0	0	0	0	0
75	0	0	0	0	0	0
76	0	0	0	0	0	0
77	0	0	0	0	0	0
78	0	0	0	0	0	0
79	0	0	0	0	0	0
80	0	0	0	0	0	0
81	0	0	0	0	0	0
82	0	0	0	0	0	0
83	0	0	0	0	0	0
84	0	0	0	0	0	0
85	0	0	0	0	0	0
86	0	0	0	0	0	0
87	0	0	0	0	0	0
88	0	0	0	0	0	0
89	0	0	0	0	0	0
90	0	0	0	0	0	0
91	0	0	0	0	0	0
92	0	0	0	0	0	0
93	0	0	0	0	0	0
94	0	0	0	0	0	0
95	0	0	0	0	0	0
96	0	0	0	0	0	0
97	0	0	0	0	0	0
98	0	0	0	0	0	0
99	0	0	0	0	0	0
100	0	0	0	0	0	0

FIG. 42

[Menu]		[Menu]	
Date: 01/01/01		Time: 12:00:00	
User: [User]		Job: [Job]	
Job Name	Job Type	Job Owner	Job ID
Job 1	Normal	Admin	10000001
Job 2	Normal	Admin	10000002
Job 3	Normal	Admin	10000003
Job 4	Normal	Admin	10000004
Job 5	Normal	Admin	10000005
Job 6	Normal	Admin	10000006
Job 7	Normal	Admin	10000007
Job 8	Normal	Admin	10000008
Job 9	Normal	Admin	10000009
Job 10	Normal	Admin	10000010
Job 11	Normal	Admin	10000011
Job 12	Normal	Admin	10000012
Job 13	Normal	Admin	10000013
Job 14	Normal	Admin	10000014
Job 15	Normal	Admin	10000015
Job 16	Normal	Admin	10000016
Job 17	Normal	Admin	10000017
Job 18	Normal	Admin	10000018
Job 19	Normal	Admin	10000019
Job 20	Normal	Admin	10000020
Job 21	Normal	Admin	10000021
Job 22	Normal	Admin	10000022
Job 23	Normal	Admin	10000023
Job 24	Normal	Admin	10000024
Job 25	Normal	Admin	10000025
Job 26	Normal	Admin	10000026
Job 27	Normal	Admin	10000027
Job 28	Normal	Admin	10000028
Job 29	Normal	Admin	10000029
Job 30	Normal	Admin	10000030
Job 31	Normal	Admin	10000031
Job 32	Normal	Admin	10000032
Job 33	Normal	Admin	10000033
Job 34	Normal	Admin	10000034
Job 35	Normal	Admin	10000035
Job 36	Normal	Admin	10000036
Job 37	Normal	Admin	10000037
Job 38	Normal	Admin	10000038
Job 39	Normal	Admin	10000039
Job 40	Normal	Admin	10000040
Job 41	Normal	Admin	10000041
Job 42	Normal	Admin	10000042
Job 43	Normal	Admin	10000043
Job 44	Normal	Admin	10000044
Job 45	Normal	Admin	10000045
Job 46	Normal	Admin	10000046
Job 47	Normal	Admin	10000047
Job 48	Normal	Admin	10000048
Job 49	Normal	Admin	10000049
Job 50	Normal	Admin	10000050
Job 51	Normal	Admin	10000051
Job 52	Normal	Admin	10000052
Job 53	Normal	Admin	10000053
Job 54	Normal	Admin	10000054
Job 55	Normal	Admin	10000055
Job 56	Normal	Admin	10000056
Job 57	Normal	Admin	10000057
Job 58	Normal	Admin	10000058
Job 59	Normal	Admin	10000059
Job 60	Normal	Admin	10000060
Job 61	Normal	Admin	10000061
Job 62	Normal	Admin	10000062
Job 63	Normal	Admin	10000063
Job 64	Normal	Admin	10000064
Job 65	Normal	Admin	10000065
Job 66	Normal	Admin	10000066
Job 67	Normal	Admin	10000067
Job 68	Normal	Admin	10000068
Job 69	Normal	Admin	10000069
Job 70	Normal	Admin	10000070
Job 71	Normal	Admin	10000071
Job 72	Normal	Admin	10000072
Job 73	Normal	Admin	10000073
Job 74	Normal	Admin	10000074
Job 75	Normal	Admin	10000075
Job 76	Normal	Admin	10000076
Job 77	Normal	Admin	10000077
Job 78	Normal	Admin	10000078
Job 79	Normal	Admin	10000079
Job 80	Normal	Admin	10000080
Job 81	Normal	Admin	10000081
Job 82	Normal	Admin	10000082
Job 83	Normal	Admin	10000083
Job 84	Normal	Admin	10000084
Job 85	Normal	Admin	10000085
Job 86	Normal	Admin	10000086
Job 87	Normal	Admin	10000087
Job 88	Normal	Admin	10000088
Job 89	Normal	Admin	10000089
Job 90	Normal	Admin	10000090
Job 91	Normal	Admin	10000091
Job 92	Normal	Admin	10000092
Job 93	Normal	Admin	10000093
Job 94	Normal	Admin	10000094
Job 95	Normal	Admin	10000095
Job 96	Normal	Admin	10000096
Job 97	Normal	Admin	10000097
Job 98	Normal	Admin	10000098
Job 99	Normal	Admin	10000099
Job 100	Normal	Admin	10000100

FIG. 43

[illegible]

FIG. 44

[illegible]

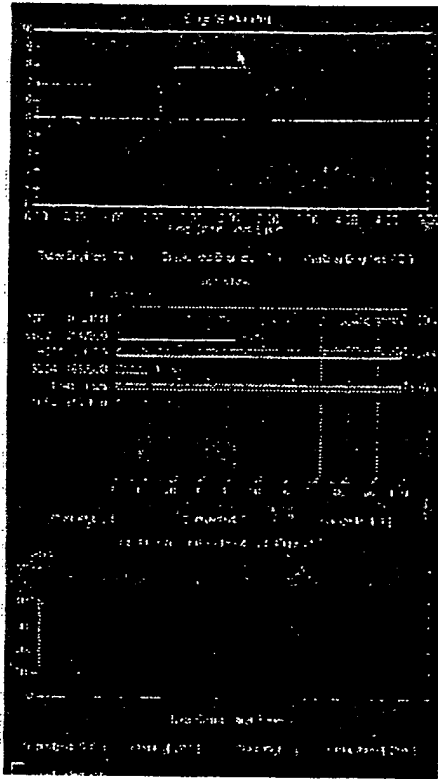
FIG. 46

The screenshot shows a Windows XP desktop with the Start menu open. The Start menu is configured to show desktop icons and favorites. The desktop background is a dark, textured image. The taskbar shows the Start button, a few open applications, and the system clock displaying 11:52 AM on 11/22/2001.

FIG. 47

TABLE 1					
Item	Hardware	Software	Hardware	Software	View
1	Hardware	0	1	1	10
2	Hardware	0	1	1	10
3	Hardware	0	1	1	10
4	Hardware	0	1	1	10
5	Hardware	0	1	1	10
6	Hardware	0	1	1	10
7	Hardware	0	1	1	10
8	Hardware	0	1	1	10
9	Hardware	0	1	1	10
10	Hardware	0	1	1	10
11	Hardware	0	1	1	10
12	Hardware	0	1	1	10
13	Hardware	0	1	1	10
14	Hardware	0	1	1	10
15	Hardware	0	1	1	10
16	Hardware	0	1	1	10
17	Hardware	0	1	1	10
18	Hardware	0	1	1	10
19	Hardware	0	1	1	10
20	Hardware	0	1	1	10
21	Hardware	0	1	1	10
22	Hardware	0	1	1	10
23	Hardware	0	1	1	10
24	Hardware	0	1	1	10
25	Hardware	0	1	1	10
26	Hardware	0	1	1	10
27	Hardware	0	1	1	10
28	Hardware	0	1	1	10
29	Hardware	0	1	1	10
30	Hardware	0	1	1	10
31	Hardware	0	1	1	10
32	Hardware	0	1	1	10
33	Hardware	0	1	1	10
34	Hardware	0	1	1	10
35	Hardware	0	1	1	10
36	Hardware	0	1	1	10
37	Hardware	0	1	1	10
38	Hardware	0	1	1	10
39	Hardware	0	1	1	10
40	Hardware	0	1	1	10
41	Hardware	0	1	1	10
42	Hardware	0	1	1	10
43	Hardware	0	1	1	10
44	Hardware	0	1	1	10
45	Hardware	0	1	1	10
46	Hardware	0	1	1	10
47	Hardware	0	1	1	10
48	Hardware	0	1	1	10
49	Hardware	0	1	1	10
50	Hardware	0	1	1	10
51	Hardware	0	1	1	10
52	Hardware	0	1	1	10
53	Hardware	0	1	1	10
54	Hardware	0	1	1	10
55	Hardware	0	1	1	10
56	Hardware	0	1	1	10
57	Hardware	0	1	1	10
58	Hardware	0	1	1	10
59	Hardware	0	1	1	10
60	Hardware	0	1	1	10
61	Hardware	0	1	1	10
62	Hardware	0	1	1	10
63	Hardware	0	1	1	10
64	Hardware	0	1	1	10
65	Hardware	0	1	1	10
66	Hardware	0	1	1	10
67	Hardware	0	1	1	10
68	Hardware	0	1	1	10
69	Hardware	0	1	1	10
70	Hardware	0	1	1	10
71	Hardware	0	1	1	10
72	Hardware	0	1	1	10
73	Hardware	0	1	1	10
74	Hardware	0	1	1	10
75	Hardware	0	1	1	10
76	Hardware	0	1	1	10
77	Hardware	0	1	1	10
78	Hardware	0	1	1	10
79	Hardware	0	1	1	10
80	Hardware	0	1	1	10
81	Hardware	0	1	1	10
82	Hardware	0	1	1	10
83	Hardware	0	1	1	10
84	Hardware	0	1	1	10
85	Hardware	0	1	1	10
86	Hardware	0	1	1	10
87	Hardware	0	1	1	10
88	Hardware	0	1	1	10
89	Hardware	0	1	1	10
90	Hardware	0	1	1	10
91	Hardware	0	1	1	10
92	Hardware	0	1	1	10
93	Hardware	0	1	1	10
94	Hardware	0	1	1	10
95	Hardware	0	1	1	10
96	Hardware	0	1	1	10
97	Hardware	0	1	1	10
98	Hardware	0	1	1	10
99	Hardware	0	1	1	10
100	Hardware	0	1	1	10

FIG. 48

**FIG. 49**

Unit Name	Unit ID	Unit Type	Unit Status	Unit Location	Unit Description	Unit Remarks
Unit 1	000001	Unit 1	Unit 1	Unit 1	Unit 1	Unit 1
Unit 2	000002	Unit 2	Unit 2	Unit 2	Unit 2	Unit 2
Unit 3	000003	Unit 3	Unit 3	Unit 3	Unit 3	Unit 3
Unit 4	000004	Unit 4	Unit 4	Unit 4	Unit 4	Unit 4
Unit 5	000005	Unit 5	Unit 5	Unit 5	Unit 5	Unit 5
Unit 6	000006	Unit 6	Unit 6	Unit 6	Unit 6	Unit 6
Unit 7	000007	Unit 7	Unit 7	Unit 7	Unit 7	Unit 7
Unit 8	000008	Unit 8	Unit 8	Unit 8	Unit 8	Unit 8
Unit 9	000009	Unit 9	Unit 9	Unit 9	Unit 9	Unit 9
Unit 10	000010	Unit 10	Unit 10	Unit 10	Unit 10	Unit 10

FIG. 50

Select the elements of interest to test:

Number of samples: 10000 Number of trials: 1000 All elements

Number of CPUs: 4 Million Values Free trials Total trials Values in 1M Values in 10M

Unsorted	1	1000000000000	115.2	41007	150760	751200
Sorted	1	1000000000000	115.2	107113	136500	201200
Unsorted	1	1000000000000	115.2	107113	136500	201200
Sorted	1	1000000000000	115.2	107113	136500	201200
Unsorted	1	1000000000000	115.2	750	150760	351160
Sorted	1	1000000000000	115.2	511127	150760	106120

FIG. 52


```
public class PiInput implements TaskInput {  
    private int _seed;  
    public PiInput(int seed) {  
        _seed = seed;  
    }  
}
```

FIG. 54

PiTaskInput.h

```
class PiTaskInput : public TaskInput {
    public: PiTaskInput() { }
    PiTaskInput(int seed) : _seed(seed) { }
    void write( ostream& strm) const;
    void read(istream& strm);
    long getSeed() { return _seed; }
    private: long _seed;
};
```

PiTaskInput.cpp

```
void PiTaskInput::write( ostream& strm) const {
    strm << _seed;
}
void PiTaskInput::read(istream& strm) {
    strm >> _seed;
}
```

FIG. 55

```
public class PiOutput implements TaskOutput {  
    public double pi;  
    public PiOutput(double pi) {  
        _pi = pi;  
    }  
}
```

FIG. 56

PiTaskOutput.h

```
class PiTaskOutput : public TaskOutput {
    public: PiTaskOutput() { }
    PiTaskOutput(double piValue) : _piValue(piValue) { }
    double getPiValue() { return _piValue; }
    void write( ostream& strm) const;
    void read(istream& strm);
    private: double _piValue;
};
```

PiTaskOutput.cpp

```
void PiTaskOutput::write( ostream& strm) const {
    strm << _piValue;
}

void PiTaskOutput::read(istream& strm) {
    strm >> _piValue;
}
```

FIG. 57


```
public class PiTasklet implements Tasklet {
    public PiTasklet(int iterations) {
        _iterations = iterations;
    }
    public TaskOutput service(TaskInput input) {
        PiInput pci = (PiInput) input;
        Random r = new Random(pci.getSeed());
        double x; double y;
        int inside = 0;
        for (int i = 0; i < _iterations; i++) {
            x = r.nextDouble();
            y = r.nextDouble();
            if (distance(x, y) < 1.) {
                inside++;
            }
        }
        double pi = inside*4./_iterations;
        return new PiOutput(pi);
    }
    private double distance(double x, double y) {
        return Math.sqrt(x*x+y*y);
    }
}
```

FIG. 58

PiTasklet.h

```

class PiTasklet : public Tasklet {
public: PiTasklet(long iterations) : _iterations(iterations)
{ }
    PiTasklet() { } TaskOutput* service(TaskInput* input);
    void write( ostream& strm) const;
    void read(istream& strm);
private: long _iterations;
};

```

PiTasklet.cpp

```

void PiTasklet::write( ostream& strm) const {
    strm << _iterations;
}

void PiTasklet::read(istream& strm) {
    strm >> _iterations;
}

TaskOutput* createTaskOutput() { return new PiTaskOutput(); }
TaskInput* createTaskInput() { return new PiTaskInput(); }
Tasklet* createTasklet() { return new PiTasklet(); }
void deleteTaskOutput(TaskOutput *taskOutput){ delete taskOutput; }
void deleteTaskInput(TaskInput *taskInput){ delete taskInput; }
void deleteTasklet(Tasklet *tasklet) { delete tasklet; }
void deleteString(string *sss){ delete sss; }
double _distance(double, double);
double urand();

```

FIG. 59A

```
TaskOutput* PiTasklet::service(TaskInput* input) {
    cout << "calling PiTasklet::service" << endl;
    PiTaskInput* piTaskInput =
dynamic_cast<PiTaskInput*>(input);
    srand(piTaskInput->getSeed()); // set seed
    double x; double y;
    int inside = 0;
    for (int i = 0; i < _iterations; i++) {
        x = urand();
        y = urand();
        if (_distance(x, y) < 1.) {
            inside++;
        }
    }
    double pi = inside*4./_iterations;
    return new PiTaskOutput(pi);
}
double _distance(double x, double y) {
    return pow((pow(x,2) + pow(y,2)), 0.5);
}
double urand() {
    return rand()/((double) RAND_MAX);
}
```

FIG. 59B

```
public class PiCalcJob extends Job {
    public PiCalcJob() {
        setTasklet(new PiCalcTasklet());
    }
    public void setIterations(int iterations) {
        mIterations = iterations;
    }
    public void setNumTasks(int numTasks) {
        mNumTasks = numTasks;
    }
    public double getPiValue() {
        return mPiTotal/mNumTasks;
    }
    protected void createTaskInputs() {
        for (int i = 0; i < mNumTasks; i++) {
            addTaskInput(new PiCalcInput(mIterations));
        }
    }
    protected void processTaskOutput(TaskOutput output) {
        PiCalcOutput piOutput = (PiCalcOutput) output;
        mPiTotal += piOutput.pi;
    }
    private double mPiTotal;
    private int mIterations;
    private int mNumTasks;
}
```

FIG. 60

PiJob.h

```

class PiJob : public Job {
public: PiJob();
    PiJob(long iterations, int numTasks);
    char* getLibraryName() { return "picalc"; }
    void setNumberOfTasks(int numTasks);
    void setIterations(int iterations);
    double getPiValue() const;
private: int _tasks;
        long _iterations;
        double _piTotal;
};

```

PiJob.cpp

```

PiJob::PiJob() : _iterations((long)1E6), _tasks(10), _piTotal(0)
{ }
PiJob::PiJob(long iterations, int tasks) :
    _iterations(iterations),
    _tasks(tasks), _piTotal(0) { }
void PiJob::createTaskInputs() {
    PiTasklet tasklet(_iterations/_tasks);
    setTasklet(tasklet);
    for (int i =0; i < _tasks; i++) {
        cout << i << endl;
        PiTaskInput input(rand()); // setting random seeds
        addTaskInput(input);
    }
}
double PiJob::getPiValue() const {
    double pi = 0;
    if (getCompletedTaskCount() != 0) {
        pi = _piTotal/getCompletedTaskCount();
    }
    return pi;
}
void PiJob::processTaskOutput(TaskOutput& output) {
    PiTaskOutput piOutput = dynamic_cast<PiTaskOutput>(output);
    cout << "PiJob::processTaskOutput, pi = ";
    cout << piOutput.getPiValue() << endl;
    _piTotal += piOutput.getPiValue();
}

```

FIG. 61

```
class Serializable {
public:
    virtual ~Serializable() {}
    virtual void write( ostream& strm) const = 0;
    virtual void read(istream& strm) = 0;
    virtual string _toString() const {
        ostream ost;
        write(ost);
        ost << ends;
        return ost.str();
    }
    virtual string * _toStringPtr() const {
        ostream ost;
        write(ost);
        ost << ends;
        return new string(ost.str());
    }
    virtual void _fromString(const string& objStr) {
        istream ist(objStr.c_str(), objStr.size());
        read(ist);
    }
};
```

FIG. 62

```
//First, implement a simple TaskInput
public class DataSetTestInput implements TaskInput {
    public DataSetTestInput(int taskNum) {
        _taskNum = taskNum;
    }
    public int getTaskNum() {
        return _taskNum;
    }
    private int _taskNum;

    //Create the TaskDataSet, and add ten inputs.
    TaskDataSet tds = new TaskDataSet("DataSetTestJob");
    for (int i = 0; i < 10; i++) {
        DataSetTestInput dsio = new DataSetTestInput(i);
        tds.addTaskInput(dsio);
    }
    tds.doneSubmitting();
}
```

FIG. 63

```
//Implement the TaskOutput
public class DataSetTestOutput implements TaskOutput{
    public DataSetTestOutput(String result) {
        _result = result;
    }
    public String getResult() {
        return _result;
    }
    private String _result;
}
//Implement the DataSetJob
public class DataSetTestJob extends DataSetJob {
    public void processTaskOutput( TaskOutput out ) {

        System.out.println(((DataSetTestOutput)out).getResult());
    }
}

//Implement the Tasklet
public class DataSetTestTasklet implements Tasklet {
    public DataSetTestTasklet() {}
    public TaskOutput service( TaskInput input ){
        DataSetTestInput in = (DataSetTestInput) input;
        msg = msg + in.getTaskNum();
        System.out.println(msg);
        return new DataSetTestOutput(msg);
    }
}

// Create a job, and attach it to the set
job = new DataSetTestJob();
job.setTaskDataSet(TaskDataSet.getDataSet("DataSetTestJob"));

// set the tasklet for this file.
job.setTasklet(new DataSetTestTasklet());
job.execute();
```

FIG. 64


```

//Create a StreamTasklet that swaps input for output
public class StreamTaskletTest extends StreamTasklet {
    public void service( InputStream input, OutputStream
output){
        byte[] buf = new byte[1024];
        while (true) {
            int bytesRead = input.read(buf);
            if (bytesRead == -1) break;
            output.write(buf, 0, bytesRead);
        }
    }
}

public class StreamJobTest extends StreamJob {
    public StreamJobTest(int numTasks) {
        setTasklet(new StreamTaskletTest());
        _numTasks = numTasks;
    }
    // create task input by getting a stream in which to write
your data
    // when the stream is closed, the input is submitted for
processing
    protected void createTaskInputs() {
        for (int i = 0; i < _numTasks; i++) {
            OutputStream os = createTaskInput();
            String msg = "Task #" + i;
            os.write(msg.getBytes());
            os.close();
        }
    }
    protected void processTaskOutput( InputStream out ) {
        StreamUtil.copy(out, System.out);
    }
    private int _numTasks;
}

// Run the StreamJob.
public class Test {
    public static void main(String args[]) {
        job = new StreamJobTest(10);
        job.executeInThread();
    }
    private static Job job;
}

```

FIG. 65

```
// start db server
Properties p = new Properties();
p.load(new FileInputStream("./properties/sqltest.properties"));
Server s = new Server();
s.start(p);
int tasks = Integer.parseInt( p.getProperty( "tasks" ) );
String query = p.getProperty( "query" );
```

FIG. 66

```
# properties for sql data set test
tasks=10
query=select * from people
# database server properties ( local config )
port=2034
database=db/db
silent=false
trace=true
# database client properties ( server config )
driver=org.hsql.jdbcDriver
url=jdbc:HypersonicSQL:hsql://your-host-name:2034
user=sa
pass=
```

FIG. 67

```
//Implement DataSetJob to print a line of output
public class SQLDataSetTestJob extends DataSetJob {
    protected void processTaskOutput( TaskOutput output ) {
        System.out.println(((SQLDataSetOutput)
output).getData());
    }
}
```

FIG. 68

```
//Implement a SQLTasklet
public class SQLDataSetTestTasklet extends SQLTasklet {
    public TaskOutput service( java.sql.ResultSet input ) {
        String data = "";
        while ( input.next() ) {
            int cols = input.getMetaData().getColumnCount();
            for ( int i=1; i <= cols; i++ ) {
                data += input.getObject( i ).toString() + "
";
            }
            data += "\n";
        }
        return new SQLDataSetOutput(data);
    }
    return new SQLDataSetOutput();
}
```

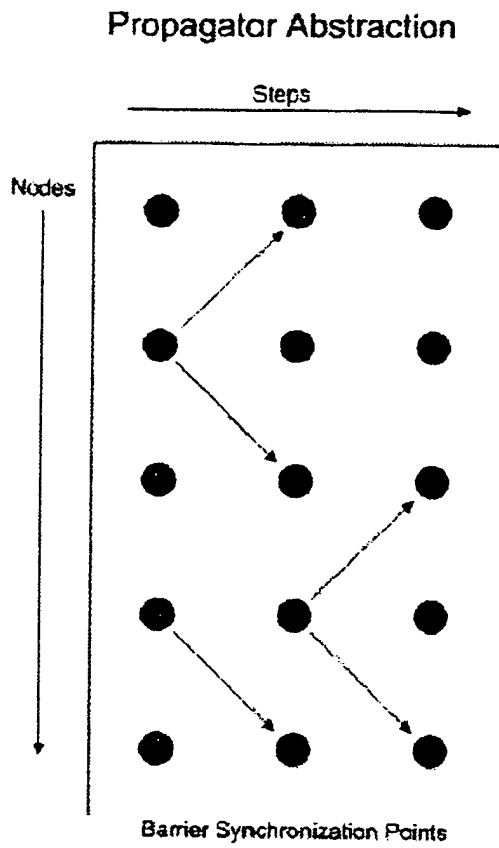
FIG. 69

```

//Implement a simple TaskOutput
public class SQLDataSetOutput implements TaskOutput {
    public SQLDataSetOutput(String s) {
        _s = s;
    }
    public String getData() {
        return _s;
    }
    private String _s;
}
public class Test {
    public static void main(String[] args) {
        if ( SQLDataSet.getDataSet( "DBTest" ) == null ) {
            // create the dataset on the server
            SQLDataSet ds = new SQLDataSet( "DBTest" );
            ds.setJDBCProperties( p );
            ds.setMode( SQLDataSet.TOTAL_TASKS, tasks );
            ds.setQuery( query );
            ds.prepare();
            while ( !ds.ready() ) {
                Thread.currentThread().sleep( 1000 );
                System.out.print( ds.getPreparedInputs() );
            }
            // now run a job on this set
            SQLDataSetTestJob job = new SQLDataSetTestJob();
            job.setTaskDataSet( SQLDataSet.getDataSet( "DBTest" )
);
            SQLDataSetTestTasklet t = new SQLDataSetTestTasklet();
            job.setTasklet( t );
            job.executeInThread();
        }
        private static SQLDataSetTestJob job;
    }
}

```

FIG. 70

**FIG. 71**

```
package examples.heatpropagator;
import com.livecluster.tasklet.propagator.*;
import java.util.*;
import java.io.*;

public class Test {
    public static void main( String[] args ) {
```

FIG. 72A


```
try {
    Properties props = new Properties();
    props.load( new FileInputStream(
"./properties/heatpropagator.properties" ) );
    int nodes = Integer.parseInt( props.getProperty(
"nodes", "1" ) );
    int cols = Integer.parseInt( props.getProperty(
"cols", "4" ) );
    int rows = Integer.parseInt( props.getProperty(
"rows", "4" ) );
    int iters = Integer.parseInt( props.getProperty(
"iters", "100" ) );
    double maxTemp = Double.parseDouble(
props.getProperty( "maxTemp", "100" ) );
    double rightBorderTemp = Double.parseDouble(
props.getProperty( "rightBorder
Temp", "100" ) );
    double leftBorderTemp = Double.parseDouble(
props.getProperty(
"leftBorderTemp", "20" ) );
    double topBorderTemp = Double.parseDouble(
props.getProperty(
"topBorderTemp", "30" ) );
    double bottomBorderTemp = Double.parseDouble(
props.getProperty(
"bottomBorderTemp", "40" ) );
    boolean computeError = new Boolean(
props.getProperty(
"computeError", "false" ) ).booleanValue();
```

FIG. 72B

```
HeatEqnSolver solver = new HeatEqnSolver( nodes,
cols, rows, iters, maxTemp,
    rightBorderTemp, leftBorderTemp,
    topBorderTemp, bottomBorderTemp,
    computeError );
System.out.println( "Running 2D heat equation with
propagation nodes: " +nodes +
    " and matrix size : " + cols + " x " + rows +
    " for " + iters + " iterations" );
```

FIG. 72C

```
        solver.solve();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
    System.exit(0);  
}  
}
```

FIG. 72D

```
package examples.heatpropagator;
import com.livecluster.tasklet.propagator.*;
import java.util.*;
import java.io.*;
public class HeatEqnSolver {
    public HeatEqnSolver( int nodes, int cols, int rows, int
iters, double maxT,
        double rightT, double leftT, double topT, double bottomT,
        boolean computeError ) {
        _nodes = nodes;
        _cols = cols;
        _rows = rows;
        _iters = iters;
        _maxTemp = maxT;
        _rightBorderTemp = rightT;
        _leftBorderTemp = leftT;
        _topBorderTemp = topT;
        _bottomBorderTemp = bottomT;
        _computeError = computeError;
    }
    public void solve() throws Exception {
        // initialize test matrix and constants
        double[][] testMatrix = new double[_rows+2][_cols+2];
        double[][] resultMatrix = new double[_rows+2][];
        double dx = 1.0 / (double)_cols;
        double dy = 1.0 / (double)_rows;
        double stabf = 0.1;
        double dt = 0.000020;
        double facx = dt * stabf / ( dx * dx );
        double facy = dt * stabf / ( dy * dy );
        GroupPropagator gp = null;
    }
}
```

FIG. 73A

```
        try {  
            gp = new GroupPropagator( "heat2d", _nodes );  
            GroupCommunicator gc = gp.getGroupCommunicator();  
            gp.setNodePropagator( new HeatPropagator( _iters,  
facx, facy ) );  
            gp.getOptions().setJarFile( new File(  
"./jars/heatpropagator.jar" ) );  
        }
```

FIG. 73B

```
        initializeMatrix( testMatrix );
        int rowsPerNode = _rows / _nodes;
        for ( int i=0, masterIndex=0; i < _nodes; i++ ) {
            if ( i == _nodes-1 ) { // for last node add
extra rows
                rowsPerNode += _rows % _nodes;
            }
            double[][] data = new
double[2*(rowsPerNode+2)][];
            for ( int j=0; j < rowsPerNode+2; j++,
masterIndex++ ) {
                data[j] = testMatrix[masterIndex];
            }
            masterIndex -= 2;
            gc.setNodeState( i, 0, data, true );
        }
```

FIG. 73C

```
        // get results
        for ( int i=0, masterIndex=1; i < _nodes; i++ ) {
            double[][] matrix =
(double[][])gc.getNodeState( i, _iters );
            for ( int j=1; j < matrix.length/2-1; j++,
masterIndex++ ) {
                resultMatrix[masterIndex] = matrix[j];
            }
        }
        System.out.println( "\nDone with 2d propagation"
);
    }
    } finally {
        if (gp != null) {
            gp.endSession();
        }
    }
}
```

FIG. 73D

```

private void initializeMatrix( double[][] testMatrix ) {
    // fill matrix with random values and set boundary values
    Random rnd = new Random();
    for ( int i=1; i < testMatrix.length-1; i++ ) {
        for ( int j=0; j < testMatrix[0].length; j++ ) {
            if ( j == 0 ) {
                testMatrix[i][j] = _leftBorderTemp;
            } else if ( j == testMatrix[0].length-1 ) {
                testMatrix[i][j] = _rightBorderTemp;
            } else {
                testMatrix[i][j] = rnd.nextDouble() *
_maxTemp;
            }
        }
    }
    for ( int j=0; j < testMatrix[0].length; j++ ) {
        testMatrix[0][j] = _topBorderTemp;
        testMatrix[testMatrix.length-1][j] = _bottomBorderTemp;
    }
}

private int _nodes, _cols, _rows, _iters;
private double _maxTemp, _rightBorderTemp, _leftBorderTemp;
private double _topBorderTemp, _bottomBorderTemp;
private boolean _computeError;

```

FIG. 73E


```
package examples.heatpropagator;
import com.livecluster.tasklet.propagator.*;
import java.io.Serializable;
public class HeatPropagator extends NodePropagator {
    public HeatPropagator( int lastIter, double facx, double
    facy ) {
        _lastIteration = lastIter;
        _facx = facx;
        _facy = facy;
    }
}
```

FIG. 74

```
    public Object propagate( int nodeId, int stepId,  
GroupCommunicator gc )  
throws Throwable {  
    double[][] matrix =(double[][])gc.getNodeState();  
    int rows = matrix.length / 2;  
    int cols = matrix[0].length;  
    .
```

FIG. 75A

```
// get boundaries from last step
if ( stepId != 0 ) {
    if ( nodeId != 0 ) {
        matrix[0] = (double[])( gc.getMessageFromSender(
nodeId-1 ) [0] );
    }
    if ( nodeId != gc.getNumNodes()-1 ) {
        matrix[rows-1] = (double[])(
gc.getMessageFromSender( nodeId+1 ) [0] );
    }
} else {
    for ( int i=1; i < matrix.length; i++ ) {
        if ( matrix[i] == null ) {
            matrix[i] = new double[matrix[0].length];
        }
    }
}
```

FIG. 75B

```
    // main loop
    for ( int i=1; i < rows-1; i++ ) {
        for ( int j=1; j < cols-1; j++ ) {
            matrix[i+rows][j] = matrix[i][j] +
                _facx * ( matrix[i+1][j] - 2.0 * matrix[i][j] +
matrix[i-1][j]) +
                _facy * ( matrix[i][j+1] - 2.0 * matrix[i][j] +
matrix[i][j-1]
                );
        }
    }
    // copy back to matrix
    for ( int i=1; i < rows-1; i++ ) {
        for ( int j=1; j < cols-1; j++ ) {
            matrix[i][j] = matrix[i+rows][j];
        }
    }
}
```

FIG. 75C

```
        // set state for next iteration
        gc.setNodeState( matrix, false );
        // send boundaries
        if ( stepId != _lastIteration ) {
            if ( nodeId > 0 ) { // bottom boundary
                gc.sendMessage( nodeId-1, matrix[1] );
            }
            if ( nodeId < gc.getNumNodes()-1 ) { // top
boundary          gc.sendMessage( nodeId+1, matrix[rows-2] );
            }
        }
        return null;
    }
    private int _lastIteration;
    private double _facx, _facy;
}
```

FIG. 75D

In XML:

```
<job class="examples.linpack.TestJob">
  <options class="com.livecluster.tasklet.JobOptions">
    <property name="jarFile" value="./jars/test.jar"/>
    <property name="serverTimeout" value="60"/>
    <property name="resubmitOnServerTimeout" value="true"/>
    <property name="priority" value="5"/>
    <property name="parallelCollection" value="false"/>
    <discriminator
      class="com.livecluster.tasklet.discriminator.BasicEngineDiscrimin
      ator">
      <property name="os" value="Linux"/>
    </discriminator>
  </options>
</job>
```

In Java:

```
Properties props = new Properties();
props.setProperty( "UserName.equals", "Bill" );
PropertyDiscriminator discrim = new PropertyDiscriminator( props
);
job.getOptions().setDiscriminator( discrim );
job.execute();
```

FIG. 76

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "NativeWrapper.h"
double calculatePi(int iters);
double distance(double x, double y);
double randm();
int main(int argc, char *argv) {
    printf("Pi = %f\n", calculatePi(10000));
    return 0;
}
JNIEXPORT jdouble JNICALL Java_NativeWrapper_calculatePi(
    *env, jobject
    thisObj, jint iterations) {
    return calculatePi(iterations);
}
double calculatePi(int iters) {
    int i;
    int inside = 0;
    double x;
    double y;
    double pi;
    static int seed = 0;
    if (!seed) {
        srand(time(NULL));
        seed = 1;
    }
    for (i = 0; i < iters; i++) {
        x = randm();
        y = randm();
        if (distance(x, y) < 1.0) {
            inside++;
        }
    }
    pi = inside*4.0/iters;
    return pi;
}
double randm() {
    return (((double) rand())/RAND_MAX);
}
double distance(double x, double y) {
    return (sqrt(x*x+y*y));
}

```

FIG. 77

```
public class NativeWrapper {
    private static boolean libLoaded = false;
    private static native double calculatePi(int iters);
    public static double getPICalc(int iters) {
        if (!libLoaded) {
            try {
                System.loadLibrary("PiCalcNative");
                libLoaded = true;
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return NativeWrapper.calculatePi(iters);
    }
}
```

FIG. 78


```
import com.LiveCluster.tasklet.*;
public class PiCalcTasklet implements Tasklet {
    private native double PiCalc(int iters);
    public PiCalcTasklet() {}
    public TaskOutput service(TaskInput input) {
        PiCalcInput pci = (PiCalcInput) input;
        double pi = NativeWrapper.getPiCalc(pci.iterations);
        return new PiCalcOutput(pi);
    }
}
```

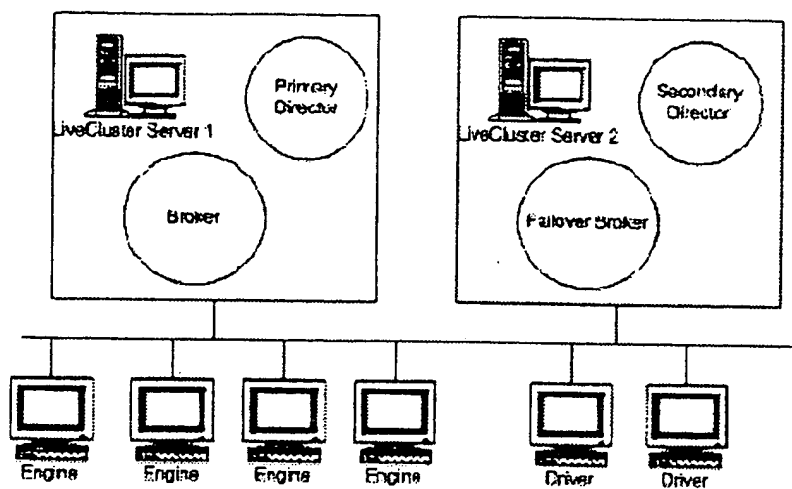
FIG. 79

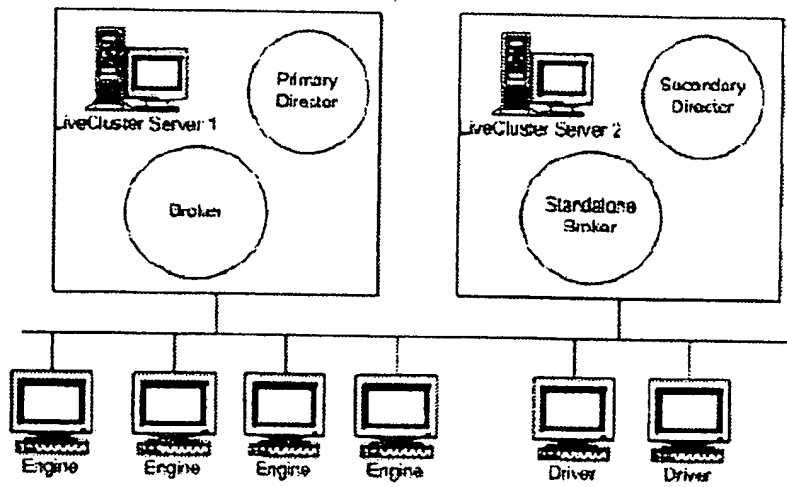
```
<?xml version="1.0" ?>
<job class=PiCalcJob>
  <property name=jarFile value=picalc.jar>
  <property name=iterations value=1000000>
  <property name=numTasks value=100>
</job>
```

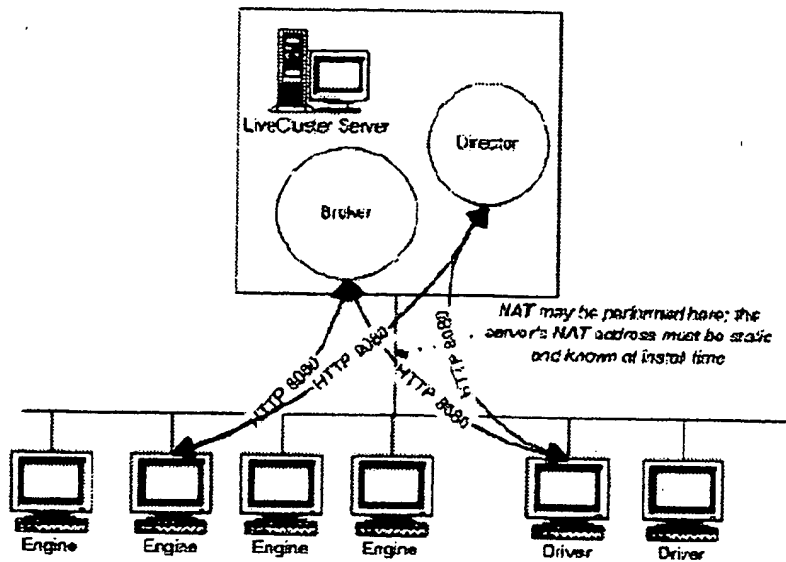
FIG. 80

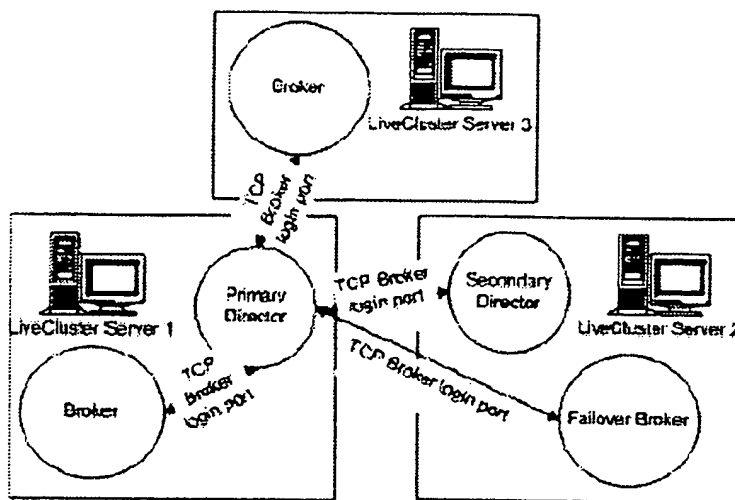
```
<batch name="Specific date serial batch test">
<property name="type" value="parallel"/>
<schedule>
<property name="type" value="absolute"/>
<property name="startTime" value="9/28/2001 11:20 AM"/>
</schedule>
<job class="examples.linpack.LinpackJob">
<property name="numberOfTasks" value="10"/>
<property name="inputSize" value="10"/>
<property name="exitOnCompleted" value="false"/>
<property name="outputSize" value="10"/>
<property name="duration" value="1"/>
<property name="cpuIntensity" value="0.1"/>
<options class="com.livecluster.tasklet.JobOptions">
<property name="username" value="batchtestuser"/>
</options>
</job>
<command class="com.livecluster.batch.command.LogCommand">
<property name="message" value="absolute batch message"/>
</command>
</batch>
```

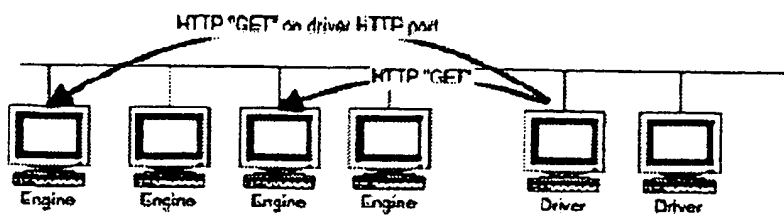
FIG. 81

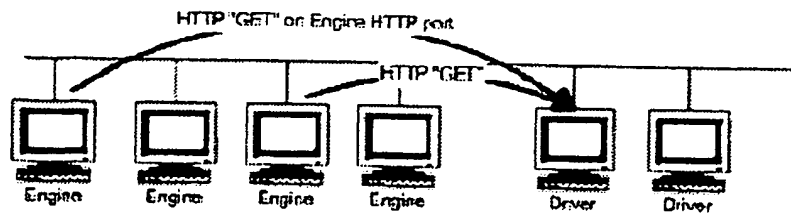
**FIG. 82**

**FIG. 83**

**FIG. 84**

**FIG. 85**

**FIG. 86**

**FIG. 87**

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US02/03218

A. CLASSIFICATION OF SUBJECT MATTER

IPC(7) : G06F 13/00, 11/32

US CL : 709/220, 222, 236, 250; 707/7

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 709/220, 222, 236, 250; 707/7

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
MICROSOFT COMPUTER DICTIONARY

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

STN, IEEE, WEST

search terms: client, server, broker, network, peer-to-peer, driver, task management, internet, time scheduler.

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y, P	US 6,304,910 B1(Roach et al) 16 October 2001, abstract, figs.3, 5A, col.4 line 10 to col.6 line 64, col.7 lines 15-58 and col.9 lines 40-65.	1-12
Y	US 5,548,726 A (Pettus) 20 August 1996, abstract, fig.1, col.2 lines 5-48, col.5 line 8 to col.6 line 53, col.8 line 19 to col.9 line 63.	1-12
Y	US 5,819,033 A (Caccvale) 06 October 1998, see abstract, fig.2, col.3 lines 12-57, col.5 line 36 to col.6 line 64.	1-12

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

* Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A" document defining the general state of the art which is not considered to be of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E" earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"I" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"Z"	document member of the same patent family
"O" document referring to an oral disclosure, use, exhibition or other means		
"P" document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search

30 MAY 2002

Date of mailing of the international search report

25 JUN 2002

Name and mailing address of the ISA/US
Commissioner of Patents and Trademarks
Box PCT
Washington, D.C. 20231

Facsimile No. (703) 305-3230

Authorized officer

AYAZ R. SHEIKH

Telephone No. (703) 305-3648